
PyBaMM Documentation

Valentin Sulzer

Mar 12, 2021

Contents

| | | |
|----------|--|------------|
| 1 | Contents | 3 |
| 1.1 | Expression Tree | 3 |
| 1.2 | Models | 23 |
| 1.3 | Parameters | 65 |
| 1.4 | Geometry | 68 |
| 1.5 | Meshes | 70 |
| 1.6 | Discretisation and spatial methods | 71 |
| 1.7 | Solvers | 84 |
| 1.8 | Post-Process Variables | 90 |
| 1.9 | Utility functions | 91 |
| 2 | Examples | 93 |
| 3 | Contributing | 95 |
| 3.1 | Adding Parameter Values | 95 |
| 3.2 | Adding a Model | 97 |
| 3.3 | Adding a Spatial Method | 100 |
| 3.4 | Adding a Solver | 102 |
| | Python Module Index | 105 |
| | Index | 107 |

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](#)

1.1 Expression Tree

1.1.1 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)

__abs__ ()
return an *AbsoluteValue* object

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

__getitem__ (*key*)
return a *Index* object

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

__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

children

returns the cached children of this node.

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

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, 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 to evaluate 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()`

Evaluates the expression. If a node exists in the tree that cannot be evaluated as a scalar or vectr (e.g. *Parameter*, *Variable*, *StateVector*), then None 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 *TypeError* is raised. !Not to be confused with `isinstance(self, pybamm.Scalar)`!

See also:

`evaluate()` evaluate the expression

`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*

See also:

`evaluate()` evaluate the expression

`jac(variable)`

Differentiate a symbol with respect to a (slice of) a State Vector. Default behaviour is to return *I* if differentiating with respect to yourself and zero otherwise. Binary and Unary Operators override this.

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

`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)

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

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”

1.1.2 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

evaluate_for_shape ()

Returns the scalar ‘NaN’ to represent the shape of a parameter. See `pybamm.Symbol.evaluate_for_shape()`

new_copy ()

See `pybamm.Symbol.new_copy()`.

class `pybamm.FunctionParameter` (*name*, **children*, *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
- **child** (*Symbol*) – child node
- **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()`.

evaluate_for_shape ()

Returns the sum of the evaluated children See `pybamm.Symbol.evaluate_for_shape()`

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()`

1.1.3 Variable

class `pybamm.Variable` (*name*, *domain=None*, *auxiliary_domains=None*)

A node in the expression tree representing a dependent 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** –

evaluate_for_shape ()

See `pybamm.Symbol.evaluate_for_shape_using_domain()`

new_copy ()

See `pybamm.Symbol.new_copy()`.

1.1.4 Independent Variable

class `pybamm.IndependentVariable` (*name*, *domain=[]*)

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** –

evaluate_for_shape ()

See `pybamm.Symbol.evaluate_for_shape_using_domain()`

class `pybamm.Time`

A node in the expression tree representing time

Extends: `Symbol`

evaluate_for_shape ()

Return the scalar '0' to represent the shape of the independent variable *Time*. See `pybamm.Symbol.evaluate_for_shape()`

new_copy ()

See `pybamm.Symbol.new_copy()`.

class `pybamm.SpatialVariable` (*name*, *domain=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*

1.1.5 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

jac (*variable*)

See `pybamm.Symbol.jac()`.

new_copy ()

See `pybamm.Symbol.new_copy()`.

set_id ()

See `pybamm.Symbol.set_id()`.

value

the value returned by the node when evaluated

1.1.6 Matrix

class `pybamm.Matrix` (*entries*, *name=None*, *domain=[]*, *entries_string=None*)

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

Extends: *Array*

Parameters

- **entries** (*numpy.array*) – the array associated with the node
- **name** (*str*, *optional*) – the name of the node

1.1.7 Vector

class `pybamm.Vector` (*entries*, *name=None*, *domain=[]*, *entries_string=None*)

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

Extends: *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

1.1.8 State Vector

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

`evaluate_for_shape()`

Returns a vector of NaNs to represent the shape of a StateVector. The size of a StateVector is the number of True elements in its `evaluation_array` See `pybamm.Symbol.evaluate_for_shape()`

`evaluation_array`

Array to use for evaluating

`jac(variable)`

Differentiate a slice of a StateVector of size m with respect to another slice of a StateVector of size n. This returns a (sparse) matrix of size m x n with ones where the y slices match, and zeros elsewhere.

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

`new_copy()`

See `pybamm.Symbol.new_copy()`.

`set_evaluation_array(y_slices)`

Set evaluation array using slices

`set_id()`

See `pybamm.Symbol.set_id()`

size

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

1.1.9 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, known_evals=None*)
See `pybamm.Symbol.evaluate()`.

evaluate_for_shape ()
See `pybamm.Symbol.evaluate_for_shape()`.

evaluates_on_edges ()
See `pybamm.Symbol.evaluates_on_edges()`.

get_children_auxiliary_domains (*l_aux_domains, r_aux_domains*)
Combine auxiliary domains from children, at all levels

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 $i = \text{grad}(\phi)$) 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.Outer* (*left, right*)

A node in the expression tree representing an outer product. This takes a 1D vector in the current collector domain of size (n,1) and a 1D variable of size (m,1), takes their outer product, and reshapes this into a vector of size (nm,1). It can also take in a vector in a single particle and a vector of the electrolyte domain to repeat that particle. Note: this class might be a bit dangerous, so at the moment it is very restrictive in what symbols can be passed to it

Extends: *BinaryOperator*

diff (*variable*)

See *pybamm.Symbol.diff()*.

class *pybamm.Kron* (*left, right*)

A node in the expression tree representing a (sparse) kronecker product operator

Extends: *BinaryOperator*

diff (*variable*)

See *pybamm.Symbol.diff()*.

jac (*variable*)

See *pybamm.Symbol.jac()*.

pybamm.outer (*left, right*)

Return outer product of two symbols. If the symbols have the same domain, the outer product is just a multiplication. If they have different domains, make a copy of the left child with same domain as right child, and then take outer product.

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

A convenience function for creating (part of) an expression tree representing a source term. This is necessary for spatial methods where the mass matrix is not the identity (e.g. finite element formulation with piecewise 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 accounting 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 by 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 be 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.

1.1.10 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, known_evals=None*)

See *pybamm.Symbol.evaluate()*.

evaluate_for_shape ()

Default behaviour: unary operator has same shape as child See *pybamm.Symbol.evaluate_for_shape()*

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()*.

jac (*variable*)

See *pybamm.Symbol.jac()*.

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

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

evaluate_for_shape ()

Default behaviour: unary operator has same shape as child See *pybamm.Symbol.evaluate_for_shape()*

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()*.

jac (*variable*)

See *pybamm.Symbol.jac()*.

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 implemented 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 formulation where we only require the (scalar 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 Dirichlet boundary conditions where necessary (e.g. in the finite element formulation) **Extends:** *SpatialOperator*

evaluate_for_shape ()

Default behaviour: unary operator has same shape as child See *pybamm.Symbol.evaluate_for_shape()*

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 $u \in \text{domain}$. 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*) –

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.IndefiniteIntegral` (*child, integration_variable*)

A node in the expression tree representing an indefinite integral operator

$$I = \int_{x_{extmin}}^x f(u) du$$

where $u \in \text{domain}$ 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** (** *Integral*) –

evaluate_for_shape()

See `pybamm.Symbol.evaluate_for_shape_using_domain()`

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*) –

evaluate_for_shape()

See `pybamm.Symbol.evaluate_for_shape_using_domain()`

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*) –

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.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*) –

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*) –

evaluate_for_shape()

See `pybamm.Symbol.evaluate_for_shape_using_domain()`

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*, *set_domain=False*)

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.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*

1.1.11 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, known_evals=None*)

See *pybamm.Symbol.evaluate()*.

evaluate_for_shape ()

See *pybamm.Symbol.evaluate_for_shape()*

get_children_auxiliary_domains (*children*)

Combine auxiliary domains from children, at all levels

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, 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

- **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

1.1.12 Broadcasting Operators

class `pybamm.Broadcast` (*child*, *broadcast_domain*, *auxiliary_domains*=None, *broadcast_type*='full', *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.

Parameters

- **child** (`Symbol`) – child node
- **broadcast_domain** (iterable of `str`) – Primary domain for broadcast. This will become the domain of the symbol
- **auxiliary_domain** (iterable of `str`) – Secondary domain for broadcast. Currently, this is only used for testing that symbols have the right shape.
- **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`) –

check_and_set_domain_and_broadcast_type (*child*, *broadcast_domain*, *broadcast_type*)

Set broadcast domain and broadcast type, performing basic checks to make sure it is compatible with the child

evaluate_for_shape ()

Returns a vector of NaNs to represent the shape of a Broadcast. See `pybamm.Symbol.evaluate_for_shape_using_domain` ()

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

A class for full broadcasts

evaluate_for_shape ()

Returns a vector of NaNs to represent the shape of a Broadcast. See `pybamm.Symbol.evaluate_for_shape_using_domain` ()

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

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*) –

evaluate_for_shape()

Returns a vector of NaNs to represent the shape of a Broadcast. See `pybamm.Symbol.evaluate_for_shape_using_domain()`

1.1.13 Simplify

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

simplify(*symbol*)

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*) –
- **symbol to simplify**(*The*) –

Returns

- *pybamm.Symbol*
- *Simplified symbol*

`pybamm.simplify_if_constant`(*symbol*)

Utility function to simplify an expression tree if it evaluates 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

1.1.14 Functions

class `pybamm.Function` (*function, *children*)

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)` returns `func(child0.evaluate(t, y), child1.evaluate(t, y), etc)`.
- **children** (*`pybamm.Symbol`*) – The children nodes to apply the function to
- ****Extends** (*** `pybamm.Symbol`*) –

diff (*variable*)

See `pybamm.Symbol.diff()`.

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

See `pybamm.Symbol.evaluate()`.

evaluate_for_shape ()

Default behaviour: has same shape as all child See `pybamm.Symbol.evaluate_for_shape()`

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*)
Returns logarithmic function of child.

`pybamm.max` (*child*)
Returns max function of child.

`pybamm.min` (*child*)
Returns min function of child.

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.

1.1.15 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, known_evals=None*)
Acts as a drop-in replacement for *pybamm.Symbol.evaluate()*

1.2 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.

1.2.1 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 that should cause the solver to terminate (e.g. concentration goes negative)

Type `list`

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`

jacobian

Contains the Jacobian for the model. If `model.use_jacobian` is `True`, the Jacobian is computed automatically during the set up in `solve`

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 tree representing the rhs and algebraic equations, Jacobian (if using) and events, before solving the model (default is `True`)

Type `bool`

use_to_python

Whether to convert the expression tree representing the rhs and algebraic equations, Jacobian (if using) and events into pure python code that will calculate the result of calling `evaluate(t, y)` on the given expression tree (default is `True`)

Type `bool`

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 ()

Check 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

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”. Must be ‘False’ for lithium-ion models.
- **“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”, “potential pair quite conductive” or “single particle potential pair”.
- **“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), “x-full”, “x-lumped”, “xyz-lumped” or “lumped”. Must be “isothermal” for lead-acid models.
- **“thermal current collector”** [bool, optional] Whether to include thermal effects in the current collector in one-dimensional models (default is False). Note that this option only takes effect if “dimensionality” is 0. If “dimensionality” is 1 or 2 current collector effects are always included. Must be ‘False’ for lead-acid models.

Type `dict`

Extends: `pybamm.BaseModel`

default_solver

Create and return the default solver for this model

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 discretisation to use

Returns Processed symbol

Return type *pybamm.Symbol*

set_soc_variables()

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

1.2.2 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'*)
Single Particle Model (SPM) of a lithium-ion battery, from¹.

References

Extends: *pybamm.lithium_ion.BaseModel*

default_solver

Create and return the default solver for this model

Single Particle Model with Electrolyte (SPMe)

class `pybamm.lithium_ion.SPMe` (*options=None, name='Single Particle Model with electrolyte'*)
Single Particle Model with Electrolyte (SPMe) of a lithium-ion battery, from¹.

References

Extends: *pybamm.lithium_ion.BaseModel*

default_solver

Create and return the default solver for this model

¹ 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).

¹ 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).

Doyle-Fuller-Newman (DFN)

class `pybamm.lithium_ion.DFN` (*options=None, name='Doyle-Fuller-Newman model'*)
Doyle-Fuller-Newman (DFN) model of a lithium-ion battery, from¹.

References

Extends: `pybamm.lithium_ion.BaseModel`

default_solver
Create and return the default solver for this model

1.2.3 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'*)
Leading-Order Quasi-Static model for lead-acid, from¹.

References

Extends: `pybamm.lead_acid.BaseModel`

default_solver
Create and return the default solver for this model

Higher-Order Models

class `pybamm.lead_acid.BaseHigherOrderModel` (*options=None, name='Composite model'*)
Base model for higher-order models for lead-acid, from¹. Uses leading-order model from `pybamm.lead_acid.LOQS`

¹ 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).

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.

References

Extends: `pybamm.lead_acid.BaseModel`

default_solver

Create and return the default solver for this model

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'*)

First-order quasi-static model for lead-acid, from¹. 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.Composite` (*options=None, name='Composite model'*)

Composite model for lead-acid, from¹. 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'*)

Extended composite model for lead-acid, from². Uses leading-order model from `pybamm.lead_acid.LOQS`

References

Extends: `pybamm.lead_acid.BaseHigherOrderModel`

set_full_porosity_submodel()

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

Full Model

class `pybamm.lead_acid.Full` (*options=None, name='Full model'*)

Porous electrode model for lead-acid, from¹, based on the Full model.

References

Extends: `pybamm.lead_acid.BaseModel`

default_solver

Create and return the default solver for this model

² V Sulzer. Mathematical modelling of lead-acid batteries. PhD thesis, University of Oxford, 2019.

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.

1.2.4 Submodels

Base Submodel

class `pybamm.BaseSubModel` (*param*, *domain=None*, *reactions=None*)

The base class for all submodels. All submodels inherit from this class and must only provide public methods which overwrite those in this base class. Any methods added to a submodel that do not overwrite those in this base 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`

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 dictionary of events that should cause the solver to terminate (e.g. concentration goes negative). The keys are strings and the values are symbols.

Type `dict`

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type `dict`

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type `dict`

set_algebraic(variables)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of `self.algebraic`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

set_boundary_conditions(variables)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

set_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 as implemented in `pybamm.BaseSubModel`.

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

set_initial_conditions(variables)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

set_rhs(variables)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of `self.rhs`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

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`

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 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.EffectiveResistance2D`

A model which calculates the effective Ohmic resistance of the current collectors in the limit of large electrical conductivity. Note: This submodel should be solved before a one-dimensional model to calculate and return the effective current collector resistance.

Extends: `pybamm.BaseModel`

get_processed_potentials (*solution, mesh, 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.

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_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type `dict`

Potential Pair models**class** `pybamm.current_collector.BasePotentialPair` (*param*)

A submodel for Ohm's law plus conservation of current in the current collectors.

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 as implemented in `pybamm.BaseSubModel`.

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

class `pybamm.current_collector.PotentialPair2plus1D` (*param*)

Base class for a 2+1D potential pair model

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

class `pybamm.current_collector.PotentialPair1plus1D` (*param*)

Base class for a 1+1D potential pair model

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of

self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

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 as 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*)

Single Particle Potential Pair models

class `pybamm.current_collector.SingleParticlePotentialPair` (*param*)

A submodel for Ohm’s law plus conservation of current in the current collectors, which uses the voltage-current relationship from the SPM(e).

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

Extends: `pybamm.current_collector.PotentialPair2plus1D`

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*

Convection

Base Model

class `pybamm.convection.BaseModel` (*param*)
Base class for convection submodels.

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

Extends: `pybamm.BaseSubModel`

No Convection

class `pybamm.convection.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.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.convection.LeadingOrder` (*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.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*

Composite Model

class `pybamm.convection.Composite` (*param*)

Class for composite pressure-driven convection

Parameters

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

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.convection.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.BaseModel`

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type *dict*

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type *dict*

set_algebraic (*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of `self.algebraic`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Electrode

Electrode Base Model

class `pybamm.electrode.BaseElectrode` (*param, domain, reactions=None*)

Base class for electrode submodels.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either ‘Negative’ or ‘Positive’
- ****Extends** (*** pybamm.BaseSubModel*) –

default_solver

Create and return the default solver for this model

Ohmic

Base Model

class `pybamm.electrode.ohm.BaseModel` (*param, domain, reactions=None*)

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`

default_solver

Create and return the default solver for this model

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Leading Order Model

class `pybamm.electrode.ohm.LeadOrder` (*param, domain*)

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'

Extends: `pybamm.electrode.ohm.BaseModel`

default_solver

Create and return the default solver for this model

get_coupled_variables (*variables*)

Returns variables which are derived from the fundamental variables in the model.

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

Composite Model

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

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`

default_solver

Create and return the default solver for this 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*

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

Full Model

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

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`

default_solver

Create and return the default solver for this model

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type *dict*

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type *dict*

set_algebraic (*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of `self.algebraic`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in `pybamm.BaseSubModel`.

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

Surface Form

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

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`

default_solver

Create and return the default solver for this 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*

Electrolyte

Base Electrolyte Conductivity Submodel

class `pybamm.electrolyte.BaseElectrolyteConductivity` (*param*, *domain=None*, *reactions=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*) –

Base Electrolyte Diffusion Submodel

class `pybamm.electrolyte.BaseElectrolyteDiffusion` (*param*, *reactions=None*)

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.

Stefan-Maxwell

Conductivity

Base Model

```
class pybamm.electrolyte.stefan_maxwell.conductivity.BaseModel (param, do-  
                                                                main=None, re-  
                                                                actions=None)
```

Base class for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*, *optional*) – The domain in which the model holds
- **reactions** (*dict*, *optional*) – Dictionary of reaction terms
- ****Extends** (** `pybamm.electrolyte.BaseElectrolyteConductivity`) –

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.stefan_maxwell.conductivity.LeadingOrder (param, do-  
                                                                main=None,  
                                                                reac-  
                                                                tions=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.BaseStefanMaxwellConductivity`) –

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.stefan_maxwell.conductivity.Composite` (*param*, *domain*, *main=None*)

Class for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations. (Composite refers to a composite leading and first-order expression from the asymptotic reduction)

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*, *optional*) – The domain in which the model holds
- ****Extends** (*** pybamm.electrolyte.stefan_maxwell.conductivity.BaseHigherOrder*) –

unpack (*variables*)

Unpack variables and return average values

Full Model

class `pybamm.electrolyte.stefan_maxwell.conductivity.Full` (*param*, *reactions*)

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.BaseStefanMaxwellConductivity*) –

default_solver

Create and return the default solver for this model

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type *dict*

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type `dict`

set_algebraic (*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of `self.algebraic`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_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 as implemented in `pybamm.BaseSubModel`.

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

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_rhs (*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of `self.rhs`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Surface Form

Full Model

```
class pybamm.electrolyte.stefan_maxwell.conductivity.surface_potential_form.FullDifferenti
```

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.stefan_maxwell.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.stefan_maxwell.conductivity.surface_potential_form.FullAlgebraic` (*pybamm.FullAlgebraicForm*)

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.stefan_maxwell.conductivity.surface_potential_form.LeadingOrderDiscretised` (*pybamm.LeadingOrderDiscretisedForm*)

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.stefan_maxwell.conductivity.surface_potential_form.LeadingOrderAlgebraic` (*pybamm.LeadingOrderAlgebraicForm*)

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.

Diffusion

Base Model

class `pybamm.electrolyte.stefan_maxwell.diffusion.BaseModel` (*param*, *reactions=None*)

Base class for conservation of mass in the electrolyte employing the Stefan-Maxwell constitutive equations.

Parameters

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

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Constant Concentration

class `pybamm.electrolyte.stefan_maxwell.diffusion.ConstantConcentration` (*param*)
Class for constant concentration of electrolyte

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

Extends: `pybamm.electrolyte.stefan_maxwell.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*

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Composite Model

class `pybamm.electrolyte.stefan_maxwell.diffusion.Composite` (*param, reactions, 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.stefan_maxwell.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

Full Model

class `pybamm.electrolyte.stefan_maxwell.diffusion.Full` (*param, reactions*)

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.stefan_maxwell.diffusion.BaseModel`) –

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type `dict`

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent

of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type `dict`

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_rhs (*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of `self.rhs`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Leading Order Model

class `pybamm.electrolyte.stefan_maxwell.diffusion.LleadingOrder` (*param*, *reac-*
tions)

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.stefan_maxwell.diffusion.BaseModel*) –

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type `dict`

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type `dict`

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_rhs (*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of `self.rhs`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Interface

Interface Base Model

class `pybamm.interface.BaseInterface` (*param, domain*)

Base class for interfacial currents

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

Extends: `pybamm.BaseSubModel`

Diffusion-limited Kinetics

Base Model

class `pybamm.interface.diffusion_limited.BaseModel` (*param, domain*)

Leading-order submodel for diffusion-limited kinetics

Parameters

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

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*

Full Model

class `pybamm.interface.diffusion_limited.FullDiffusionLimited` (*param, domain*)

Full submodel for diffusion-limited kinetics

Parameters

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

Extends: `pybamm.interface.diffusion_limited.BaseModel`

Leading-order Model

```
class pybamm.interface.diffusion_limited.LeadingOrderDiffusionLimited(param,  
                                                                    do-  
                                                                    main)
```

Leading-order submodel for diffusion-limited kinetics

Parameters

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

Extends: `pybamm.interface.diffusion_limited.BaseModel`

Inverse Interface Kinetics

Base Inverse First-order Kinetics

```
class pybamm.interface.inverse_kinetics.BaseInverseFirstOrderKinetics(param,  
                                                                    do-  
                                                                    main)
```

Base inverse first-order kinetics

Parameters

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

Extends: `pybamm.interface.kinetics.BaseFirstOrderKinetics`

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`

Base Inverse Kinetics

```
class pybamm.interface.inverse_kinetics.BaseInverseKinetics(param, domain)
```

A base 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.
- ****Extends** (** `pybamm.interface.kinetics.ButlerVolmer`) –

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type `dict`

Inverse Butler-Volmer

class `pybamm.interface.inverse_kinetics.InverseButlerVolmer` (*param, domain*)

A base 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.
- ****Extends** (** `pybamm.interface.kinetics.ButlerVolmer`) –

Interface Kinetics

Base Kinetics

class `pybamm.interface.kinetics.BaseModel` (*param, domain*)

Base submodel for kinetics

Parameters

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

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`

Base First-order Kinetics

class `pybamm.interface.kinetics.BaseFirstOrderKinetics` (*param, domain*)
Base first-order kinetics

Parameters

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

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*

Butler-Volmer

class `pybamm.interface.kinetics.ButlerVolmer` (*param, domain*)
Base submodel which implements the forward Butler-Volmer equation:

$$j = j_0(c) * \sinh(\eta_r(c))$$

Parameters

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

Extends: `pybamm.interface.kinetics.BaseModel`

class `pybamm.interface.kinetics.FirstOrderButlerVolmer` (*param, domain*)

No Reaction

class `pybamm.interface.kinetics.NoReaction` (*param, domain*)
Base submodel for when no reaction occurs

Parameters

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

Extends: `pybamm.interface.kinetics.BaseModel`

Tafel

class `pybamm.interface.kinetics.ForwardTafel` (*param*, *domain*)

Base submodel which implements the forward 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.BaseModel`

class `pybamm.interface.kinetics.FirstOrderForwardTafel` (*param*, *domain*)

class `pybamm.interface.kinetics.BackwardTafel` (*param*, *domain*)

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.BaseModel`

Lead Acid

class `pybamm.interface.lead_acid.ButlerVolmer` (*param*, *domain*)

Extends `BaseInterfaceLeadAcid` (for exchange-current density, etc) and `kinetics.ButlerVolmer` (for kinetics)

class `pybamm.interface.lead_acid.InverseButlerVolmer` (*param*, *domain*)

Extends `BaseInterfaceLeadAcid` (for exchange-current density, etc) and `inverse_kinetics.InverseButlerVolmer` (for kinetics)

class `pybamm.interface.lead_acid.FirstOrderButlerVolmer` (*param*, *domain*)

Extends `BaseInterfaceLeadAcid` (for exchange-current density, etc) and `kinetics.FirstOrderButlerVolmer` (for kinetics)

class `pybamm.interface.lead_acid.InverseFirstOrderKinetics` (*param*, *domain*)

Extends `BaseInterfaceLeadAcid` (for exchange-current density, etc) and `kinetics.BaseInverseFirstOrderKinetics` (for kinetics)

Lithium-Ion

class `pybamm.interface.lithium_ion.ButlerVolmer` (*param*, *domain*)

Extends `BaseInterfaceLithiumIon` (for exchange-current density, etc) and `kinetics.ButlerVolmer` (for kinetics)

class `pybamm.interface.lithium_ion.InverseButlerVolmer` (*param*, *domain*)

Extends `BaseInterfaceLithiumIon` (for exchange-current density, etc) and `inverse_kinetics.InverseButlerVolmer` (for kinetics)

Oxygen Diffusion

Base Model

class `pybamm.oxygen_diffusion.BaseModel` (*param, reactions=None*)
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, reactions, 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, reactions*)

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, reactions*)

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 as implemented in *pybamm.BaseSubModel*.

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

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_rhs (*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of `self.rhs`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Leading Order Model

class `pybamm.oxygen_diffusion.LeadinOrder` (*param, reactions*)

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.oxygen_diffusion.BaseModel*) –

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type *dict*

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type *dict*

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_rhs (*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of `self.rhs`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

No Oxygen

class `pybamm.oxygen_diffusion.NoOxygen` (*param*)

Class for when there is no oxygen

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

Extends: `pybamm.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.

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.

Fickian

Base Model

class `pybamm.particle.fickian.BaseModel` (*param, domain*)

Base class for molar conservation in particles which employ 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`

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_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.

Many Particle

class `pybamm.particle.fickian.ManyParticles` (*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.fickian.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`

Single Particle

class `pybamm.particle.fickian.SingleParticle` (*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.fickian.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*

Fast

Base Model

class `pybamm.particle.fast.BaseModel` (*param*, *domain*)

Base class for molar conservation in 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`

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.

Many Particle

class `pybamm.particle.fast.ManyParticles` (*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.fast.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`

Single Particle

class `pybamm.particle.fast.SingleParticle` (*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.fast.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`

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`

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.LeadinOrder` (*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 as implemented in `pybamm.BaseSubModel`.

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

set_rhs (*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of `self.rhs`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

Thermal

Isothermal

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`

X-full

Base Model

class `pybamm.thermal.x_full.BaseModel` (*param*)

Base class for full x-direction thermal submodels.

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

Extends: `pybamm.thermal.BaseModel`

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type `dict`

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type `dict`

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

No current collector

class `pybamm.thermal.x_full.NoCurrentCollector` (*param*)

Class for full x-direction thermal submodel without current collectors

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

Extends: `pybamm.thermal.x_full.BaseModel`

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_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.

X-lumped

Base Model

class `pybamm.thermal.x_lumped.BaseModel` (*param*)

Base class for x-lumped thermal submodel

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

Extends: `pybamm.thermal.BaseModel`

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type *dict*

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type `dict`

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

No current collector

class `pybamm.thermal.x_lumped.NoCurrentCollector` (*param*)

Class for x-lumped thermal submodel without current collectors. Note: since there are no current collectors in this model, the electrochemical model must be 1D (x-direction only).

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

Extends: `pybamm.thermal.BaseModel`

set_rhs (*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of `self.rhs`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

0D current collector

class `pybamm.thermal.x_lumped.CurrentCollector0D` (*param*)

Class for x-lumped thermal model with 0D current collectors

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.

1D current collector

class `pybamm.thermal.x_lumped.CurrentCollector1D` (*param*)

Class for x-lumped thermal model with 1D current collectors

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_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.

2D current collector

class `pybamm.thermal.x_lumped.CurrentCollector2D` (*param*)

Class for x-lumped thermal submodel with 2D current collectors

set_boundary_conditions (*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of `self.boundary_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_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.

XYZ-lumped

Base Model

class `pybamm.thermal.xyz_lumped.BaseModel` (*param*)

Base class for xyz-lumped thermal submodel

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

Extends: `pybamm.thermal.BaseModel`

get_coupled_variables (*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in ‘get_fundamental_variables’ instead of this method.

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

Returns The variables created in this submodel which depend on variables in other submodels.

Return type *dict*

get_fundamental_variables ()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns The variables created by the submodel which are independent of variables in other submodels.

Return type *dict*

set_initial_conditions (*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of `self.initial_conditions`. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

set_rhs (*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of ‘pass’ is used as implemented in `pybamm.BaseSubModel`.

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

1D current collector

class `pybamm.thermal.xyz_lumped.CurrentCollector1D` (*param*)

Class for xyz-lumped thermal submodel with 1D current collectors

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

Extends: `pybamm.thermal.BaseModel`

2D current collector

class `pybamm.thermal.xyz_lumped.CurrentCollector2D` (*param*)

Class for xyz-lumped thermal submodel with 2D current collectors

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

Extends: `pybamm.thermal.BaseModel`

Base Thermal

class `pybamm.thermal.BaseThermal` (*param*)

Base class for thermal effects

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

Extends: `pybamm.BaseSubModel`

1.3 Parameters

1.3.1 Base Parameter Values

class `pybamm.ParameterValues` (*base_parameters={}, optional_parameters={}*)

The parameter values for a simulation.

Parameters

- **base_parameters** (*dict or string*) – The base parameters. If string, gets passed to `read_parameters_csv` to read a file.
- **optional_parameters** (*dict or string*) – Optional parameters, overwrites `base_parameters` if there is a conflict. If string, gets passed to `read_parameters_csv` to read a file.

process_geometry (*geometry*)

Assign parameter values to a geometry (inplace).

Parameters `geometry` (`pybamm.Geometry`) – Geometry specs to assign parameter values to

process_model (`model`, `processing='process'`)

Assign parameter values to a model. Currently inplace, could be changed to return a new model.

Parameters

- **model** (`pybamm.BaseModel`) – Model to assign parameter values for
- **processing** (`str`, `optional`) – Flag to indicate how to process model (default 'process')
 - 'process': Calls `process_symbol()` (walk through the symbol and replace any Parameter with a Value)
 - 'update': Calls `update_scalars()` for use on already-processed model (update the value of any Scalars in the expression tree.)

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

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`

update (`[E]`, `**F`) \rightarrow None. Update D from dict/iterable E and F.

If E is present and has a `.keys()` method, then does: for k in E: $D[k] = E[k]$ If E is present and lacks a `.keys()` method, then does: for k, v in E: $D[k] = v$ In either case, this is followed by: for k in F: $D[k] = F[k]$

update_model (`model`, `disc`)

Process a discretised model. Currently inplace, could be changed to return a new model.

Parameters

- **model** (`pybamm.BaseModel`) – Model to assign parameter values for
- **disc** (`pybamm.Discretisation`) – The class that was used to discretise

update_scalars (`symbol`)

Update the value of any Scalars in the expression tree.

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

Returns **symbol** – Symbol with Scalars updated

Return type `pybamm.Symbol`

1.3.2 Geometric Parameters

Standard geometric parameters

1.3.3 Electrical Parameters

1.3.4 Thermal Parameters

1.3.5 Standard Lithium-ion Parameters

Standard parameters for lithium-ion battery models

1.3.6 Standard Lead-Acid Parameters

Standard Parameters for lead-acid battery models

1.3.7 Print parameters

`pybamm.print_parameters(parameters, parameter_values, 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 $x \cdot \text{Crate}$ 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
- **parameter_values** (`pybamm.ParameterValues`) – The class of parameter values
- **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

`pybamm.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

1.4 Geometry

1.4.1 Geometry

class pybamm.Geometry(*geometries, custom_geometry={})

A geometry class to store the details features of the cell geometry.

Geometry extends the class dictionary and uses the key words: “negative electrode”, “positive electrode”, etc to indicate the subdomain. Within each subdomain, there are “primary”, “secondary” or “tabs” dimensions. “primary” dimensions correspond to dimensions on which spatial operators will be applied (e.g. the gradient and divergence). In contrast, spatial operators do not act along “secondary” dimensions. This allows for multiple independent particles to be included into a model.

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": {
        "primary": {x_n: {"min": pybamm.Scalar(0), "max": l_n}}
    }
}
```

A user can create a new Geometry by combining one or more of the pre-defined geometries defined with the names given below.

- “1D macro”: macroscopic 1D cell geometry (i.e. electrodes)
- “3D macro”: macroscopic 3D cell geometry
- “1+1D macro”: 1D macroscopic cell geometry with a 1D current collector
- “1+2D macro”: 1D macroscopic cell geometry with a 2D current collector
- “1D micro”: 1D microscopic cell geometry (i.e. particles)
- “1+1D micro”: This is the geometry used in the standard DFN or P2D model
- “(1+0)+1D micro”: 0D macroscopic cell geometry with 1D current collector, along with the microscopic 1D particle geometry.
- “(2+0)+1D micro”: 0D macroscopic cell geometry with 1D current collector, along with the microscopic 1D particle geometry.
- “(1+1)+1D micro”: 1D macroscopic cell geometry, with 1D current collector model, along with the microscopic 1D particle geometry.
- “(2+1)+1D micro”: 1D macroscopic cell geometry, with 2D current collector model, along with the microscopic 1D particle geometry.
- “2D current collector”: macroscopic 2D current collector geometry

Extends: dict

Parameters

- **geometries** (one or more strings or Geometry objects. A string will be assumed to be) – one of the predefined Geometries given above
- **custom_geometry** (dict containing any extra user defined geometry) –

add_domain (*name*, *geometry*)

Add a new domain to the geometry

Parameters

- **name** (*string* giving the name of the domain)–
- **geometry** (*dict* of variables in the domain, along with the minimum and maximum) – extents (e.g. {"primary": {x_n: {"min": pybamm.Scalar(0), "max": 1_n}}})

class pybamm.**Geometry1DMacro** (*custom_geometry*={})

A geometry class to store the details features of the macroscopic 1D cell geometry.

Extends: *Geometry*

Parameters **custom_geometry** (*dict* containing any extra user defined geometry)–

class pybamm.**Geometry3DMacro** (*custom_geometry*={})

A geometry class to store the details features of the macroscopic 3D cell geometry.

Extends: *Geometry1DMacro*

Parameters **custom_geometry** (*dict* containing any extra user defined geometry)–

class pybamm.**Geometry1DMicro** (*custom_geometry*={})

A geometry class to store the details features of the microscopic 1D particle geometry.

Extends: *Geometry*

Parameters **custom_geometry** (*dict* containing any extra user defined geometry)–

class pybamm.**Geometry1p1DMicro** (*custom_geometry*={})

A geometry class to store the details features of the 1+1D cell geometry. This is the geometry used in the standard DFN or P2D model.

Extends: *Geometry*

Parameters **custom_geometry** (*dict* containing any extra user defined geometry)–

class pybamm.**Geometryxp1DMacro** (*cc_dimension*=1, *custom_geometry*={})

A geometry class to store the details features of x+1D macroscopic cell geometry, where x is the dimension of the current collector model.

Extends: *Geometry1DMacro*

Parameters

- **cc_dimension** (*int*, *optional*)– the dimension of the current collector model
- **custom_geometry** (*dict*, *optional*)– dictionary containing any extra user defined geometry

class pybamm.**Geometryxp0p1DMicro** (*cc_dimension*=1, *custom_geometry*={})

A geometry class to store the details features of x+0D macroscopic cell geometry, where x is the dimension of the current collector model, along with the microscopic 1D particle geometry.

Extends: *Geometry1DMicro*

Parameters

- **cc_dimension** (*int*, *optional*)– the dimension of the current collector model

- **custom_geometry** (*dict*, *optional*) – dictionary containing any extra user defined geometry

class pybamm.**Geometryxplp1DMicro** (*cc_dimension=1*, *custom_geometry={}*)

A geometry class to store the details features of x+1D macroscopic cell geometry, where x is the dimension of the current collector model, along with the microscopic 1D particle geometry.

Extends: *Geometry1DMicro*

Parameters

- **cc_dimension** (*int*, *optional*) – the dimension of the current collector model
- **custom_geometry** (*dict*, *optional*) – dictionary containing any extra user defined geometry

class pybamm.**Geometry2DCurrentCollector** (*custom_geometry={}*)

A geometry class to store the details features of the macroscopic 2D current collector geometry.

Extends: *Geometry*

Parameters **custom_geometry** (*dict containing any extra user defined geometry*) –

1.5 Meshes

1.5.1 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`

1.5.2 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.

class pybamm.**Uniform1DSubMesh** (*lims, npts, tabs=None*)

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
- **tabs** (*dict, optional*) – A dictionary that contains information about the size and location of the tabs

1.5.3 Scikits FEM Submeshes

class pybamm.**Scikit2DSubMesh** (*lims, npts, tabs*)

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

- **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
- **tabs** (*dict*) – A dictionary that contains information about the size and location of the tabs

on_boundary (*y, z, tab*)

A method to get the degrees of freedom corresponding to the subdomains for the tabs.

1.5.4 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 spatial variable
- **npts** (*dict, optional*) – Number of points to be used. Included for compatibility with other meshes, but ignored by this mesh class

1.6 Discretisation and spatial methods

1.6.1 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 method to be used on each domain. The keys correspond to the keys in a `pybamm.Model`

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, or an outer product (if broadcasted, variable is a multiplication with a vector of ones)

create_mass_matrix (`model`)

Creates mass matrix of the discretised model. Note that the model is assumed to be of the form $M \cdot y_{\text{dot}} = f(t, y)$, 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 The mass matrix

Return type `pybamm.Matrix`

process_boundary_conditions (`model`)

Discretise model `boundary_conditions`, also converting keys to ids

Parameters **model** (`pybamm.BaseModel`) – Model to discretise. 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.initial_conditions` or `model.variables`).

Parameters `var_eqn_dict` (*dict*) – Equations (`{variable: equation}` dict) to discretise (can be `model.rhs`, `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 discretise. Must have attributes `rhs`, `initial_conditions` and `boundary_conditions` (all dicts of `{variable: equation}`)

Returns Tuple of `processed_initial_conditions` (dict of initial conditions) and `concatenated_initial_conditions` (numpy array of concatenated initial conditions)

Return type *tuple*

process_model (*model*, *inplace=True*)

Discretise a model. Currently inplace, could be changed to return a new model.

Parameters

- **model** (*pybamm.BaseModel*) – Model to discretise. Must have attributes `rhs`, `initial_conditions` and `boundary_conditions` (all dicts of `{variable: equation}`)
- **inplace** (*bool*, *optional*) – If True, discretise the model in place. Otherwise, return a new discretised model. Default is True.

Returns `model_disc` – The discretised model. Note that if `inplace` is True, model will have also been discretised in place so `model == model_disc`. If `inplace` is False, `model != model_disc`

Return type *pybamm.BaseModel*

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

process_rhs_and_algebraic (*model*)

Discretise model equations - differential ('rhs') and algebraic.

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

Returns Tuple of `processed_rhs` (dict of processed differential equations), `processed_concatenated_rhs`, `processed_algebraic` (dict of processed algebraic equations) and `processed_concatenated_algebraic`

Return type *tuple*

process_symbol (*symbol*)

Discretise operators in model equations. If a symbol has already been discretised, the stored value is returned.

Parameters `symbol` (*pybamm.expression_tree.symbol.Symbol*) – Symbol to discretise

Returns Discretised symbol

Return type *pybamm.expression_tree.symbol.Symbol*

set_internal_boundary_conditions (*model*)

A method to set the internal boundary conditions for the submodel. These are required to properly calculate the gradient. Note: this method modifies the state of `self.boundary_conditions`.

set_variable_slices (*variables*)

Sets the slicing for variables.

Parameters **variables** (iterable of `pybamm.Variables`) – The variables for which to set slices

1.6.2 Spatial Method

class `pybamm.SpatialMethod` (*mesh*)

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*)

Returns the boundary value or flux using the appropriate expression for the spatial method. To do this, we create a sparse vector ‘bv_vector’ that extracts either the first (for side=“left”) or last (for side=“right”) point from ‘discretised_child’.

Parameters

- **symbol** (`pybamm.Symbol`) – The boundary value or flux symbol
- **discretised_child** (`pybamm.StateVector`) – The discretised variable from which to calculate the boundary value

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
- **broadcast_type** (`str`) – The type of broadcast, either: ‘primary’ or ‘full’

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 appropriate 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*)

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

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*

1.6.3 Finite Volume

class *pybamm.FiniteVolume* (*mesh*)

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 “ $2*a - y_1$ ” where y_1 is the value of the first node. Similarly for the right-hand boundary condition.

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 “ $2*a - y_1$ ” where y_1 is the value of the first node. Similarly for the right-hand boundary condition.

For Neumann bcs, for a boundary condition “ $dy/dx = b$ at the left-hand boundary”, we concatenate a ghost node to the start of the vector y with value “ $b*h + y_1$ ” where y_1 is the value of the first node and h is the mesh size. Similarly for the right-hand boundary condition.

Parameters

- **domain** (*list of strings*) – The domain of the symbol for which to add ghost nodes
- **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* (shape (n+2, n))

boundary_value_or_flux (*symbol, discretised_child*)

Uses linear 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 (*domain, 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 **domain** (*list*) – The domain(s) 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 (*domain*)

Divergence matrix for finite volumes in the appropriate domain. Equivalent to $\text{div}(N) = (N[1:] - N[:-1])/dx$

Parameters **domain** (*list*) – The domain(s) 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*)

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*)

Gradient matrix for finite volumes in the appropriate domain. Equivalent to $\text{grad}(y) = (y[1:] - y[:-1])/dx$

Parameters **domain** (*list*) – The domain(s) 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*)

Implementation of the indefinite integral operator.

indefinite_integral_matrix_edges (*domain*)

Matrix for finite-volume implementation of the indefinite integral where the integrand is evaluated on mesh edges

$$F(x) = \int_0^x f(u) du$$

The indefinite integral must satisfy the following conditions:

- $F(0) = 0$
- $f(x) = \frac{dF}{dx}$

or, in discrete form,

- $\text{BoundaryValue}(F, \text{"left"}) = 0$, 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

- $F_0 = du_{1/2} * f_{1/2}/2$
- $F_{i+1} = F_i + du * f_{i+1/2}$

Note that $f_{-1/2}$ and $f_{n+1/2}$ 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.

Parameters **domain** (*list*) – The domain(s) of integration

Returns The finite volume integral matrix for the domain

Return type `pybamm.Matrix`

indefinite_integral_matrix_nodes (*domain*)

Matrix for finite-volume implementation of the indefinite integral where the integrand is evaluated on mesh nodes. This is just a straightforward cumulative sum of the integrand

Parameters **domain** (*list*) – The domain(s) of integration

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*)

Convert a discretised symbol evaluated on the cell nodes to a discretised symbol evaluated on the cell edges. See *pybamm.FiniteVolume.shift()*

process_binary_operators (*bin_op, left, right, disc_left, disc_right*)

Discretise binary operators in model equations. Performs appropriate averaging of diffusivities if one of the children is a gradient operator, so that discretised sizes match up.

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*)

Convert a discretised symbol evaluated at edges/nodes, to a discretised symbol evaluated at nodes/edges. For now we just take the arithmetic mean, though it may be 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”)

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*

1.6.4 Scikit Finite Elements

class *pybamm.ScikitFiniteElement* (*mesh*)

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 $-\text{grad}^2 u = f$ 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 assembled finite element matrices to account for boundary conditions.

Parameters

- **M** (*scipy.sparse.coo_matrix*) – The assembled finite element matrix to adjust.
- **boundary** (*numpy.array*) – Array of the indices which correspond to the boundary.
- **zero** (*bool, optional*) – If True, the rows of M given by the indices 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*)

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 (*domain, 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

- **domain** (*list*) – 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. See `pybamm.SpatialMethod.gradient()`

gradient_squared (*symbol, discretised_symbol, boundary_conditions*)

Matrix-vector 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`

1.6.5 Zero Dimensional Spatial Method

class `pybamm.ZeroDimensionalMethod` (*mesh=None*)

A discretisation class for the zero dimensional mesh

Parameters

- **mesh** – Contains all the submeshes for discretisation
- ****Extends**** (`pybamm.SpatialMethod`) –

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.

1.7 Solvers

1.7.1 Algebraic Solvers

class `pybamm.AlgebraicSolver` (*method='lm', tol=1e-06*)

Solve a discretised model which contains only (time independent) algebraic equations using a 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) algebraic system at each time level.

Parameters

- **method** (*str, optional*) – The method to use to solve the system (default is “lm”)
- **tolerance** (*float, optional*) – The tolerance for the solver (default is 1e-6).

root (*algebraic, y0_guess, jacobian=None*)

Calculate the solution of the algebraic equations through root-finding

Parameters

- **algebraic** (*method*) – Function that takes in *y* and returns the value of the algebraic equations
- **y0_guess** (*array-like*) – Array of the user’s guess for the solution, used to initialise the root finding algorithm
- **jacobian** (*method, optional*) – A function that takes in *t* and *y* and returns the Jacobian. If *None*, the solver will approximate the Jacobian if required.

set_up (*model*)

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*

Returns

- **concatenated_algebraic** (`pybamm.Concatenation`) – Algebraic equations, which should evaluate to zero
- **jac** (`pybamm.SparseStack`) – Jacobian matrix for the differential and algebraic equations

Raises `pybamm.SolverError` – If the model contains any time derivatives, i.e. *rhs* equations (in which case an ODE or DAE solver should be used instead)

solve (*model*)

Calculate the solution of the model.

Parameters *model* (*pybamm.BaseModel*) – The model whose solution to calculate. Must only contain algebraic equations.

1.7.2 Base Solvers

class *pybamm.BaseSolver* (*method=None, tol=1e-08*)

Solve a discretised model.

Parameters *tolerance* (*float, optional*) – The tolerance for the solver (default is 1e-8).

compute_solution (*model, t_eval*)

Calculate the solution of the model at specified times. Note: this does *not* execute the solver setup.

Parameters

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes *rhs* and *initial_conditions*
- **t_eval** (*numeric type*) – The times at which to compute the solution

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*)

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*

Raises *pybamm.SolverError* – If the model contains any algebraic equations (in which case a DAE solver should be used instead)

solve (*model, t_eval*)

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 at which to compute the solution

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

step (*model, dt, npts=2*)

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

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes `rhs` and `initial_conditions`
- **dt** (*numeric type*) – The timestep 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`).

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

class `pybamm.OdeSolver` (*method=None, tol=1e-08*)

Solve a discretised model.

Parameters `tolerance` (*float, optional*) – The tolerance for the solver (default is 1e-8).

compute_solution (*model, t_eval*)

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 at which to compute the solution

integrate (*derivs, y0, t_eval, events=None, mass_matrix=None, jacobian=None*)

Solve a model defined by `dydt` with initial conditions `y0`.

Parameters

- **derivs** (*method*) – A function that takes in `t` and `y` and returns the time-derivative `dydt`
- **y0** (*numeric type*) – The initial conditions
- **t_eval** (*numeric type*) – The times at which to compute the solution
- **events** (*method, optional*) – A function that takes in `t` and `y` and returns conditions for the solver to stop
- **mass_matrix** (*array_like, optional*) – The (sparse) mass matrix for the chosen spatial method.
- **jacobian** (*method, optional*) – A function that takes in `t` and `y` and returns the Jacobian

set_up (*model*)

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`

Raises `pybamm.SolverError` – If the model contains any algebraic equations (in which case a DAE solver should be used instead)

class `pybamm.DaeSolver` (*method=None, tol=1e-08, root_method='lm', root_tol=1e-06, max_steps=1000*)

Solve a discretised model.

Parameters

- **tolerance** (*float, optional*) – The tolerance for the solver (default is 1e-8).
- **root_method** (*str, optional*) – The method to use to find initial conditions (default is “lm”)
- **tolerance** – The tolerance for the initial-condition solver (default is 1e-8).

- **max_steps** (*int, optional*) – The maximum number of steps the solver will take before terminating (default is 1000).

calculate_consistent_initial_conditions (*rhs, algebraic, y0_guess, jac=None*)

Calculate consistent initial conditions for the algebraic equations through root-finding

Parameters

- **rhs** (*method*) – Function that takes in *t* and *y* and returns the value of the differential equations
- **algebraic** (*method*) – Function that takes in *t* and *y* and returns the value of the algebraic equations
- **y0_guess** (*array-like*) – Array of the user’s guess for the initial conditions, used to initialise the root finding algorithm
- **jac** (*method*) – Function that takes in *t* and *y* and returns the value of the jacobian for the algebraic equations

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*

compute_solution (*model, t_eval*)

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 at which to compute the solution

integrate (*residuals, y0, t_eval, events=None, mass_matrix=None, jacobian=None*)

Solve a DAE model defined by residuals with initial conditions *y0*.

Parameters

- **residuals** (*method*) – A function that takes in *t*, *y* and *ydot* and returns the residuals of the equations
- **y0** (*numeric type*) – The initial conditions
- **t_eval** (*numeric type*) – The times at which to compute the solution
- **events** (*method, optional*) – A function that takes in *t* and *y* and returns conditions for the solver to stop
- **mass_matrix** (*array_like, optional*) – The (sparse) mass matrix for the chosen spatial method.
- **jacobian** (*method, optional*) – A function that takes in *t*, *y* and *ydot* and returns the Jacobian

set_up (*model*)

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*

Raises *pybamm.SolverError* – If the model contains any algebraic equations (in which case a DAE solver should be used instead)

1.7.3 Scipy Solver

class `pybamm.ScipySolver` (*method*='BDF', *tol*=1e-08)

Solve a discretised model, using `scipy.integrate.solve_ivp`.

Parameters

- **method** (*str*, *optional*) – The method to use in `solve_ivp` (default is “BDF”)
- **tolerance** (*float*, *optional*) – The tolerance for the solver (default is 1e-8). Set as the both `reltol` and `abstol` in `solve_ivp`.

integrate (*derivs*, *y0*, *t_eval*, *events*=None, *mass_matrix*=None, *jacobian*=None)

Solve a model defined by `dydt` with initial conditions `y0`.

Parameters

- **derivs** (*method*) – A function that takes in `t` (size (1,)), `y` (size (n,)) and returns the time-derivative `dydt` (size (n,))
- **y0** (`numpy.array`, size (n,)) – The initial conditions
- **t_eval** (`numpy.array`, size (k,)) – The times at which to compute the solution
- **events** (*method*, *optional*) – A function that takes in `t` and `y` and returns conditions for the solver to stop
- **mass_matrix** (*array_like*, *optional*) – The (sparse) mass matrix for the chosen spatial method.
- **jacobian** (*method*, *optional*) – A function that takes in `t` and `y` and returns the Jacobian. If None, the solver will approximate the Jacobian.

Returns An object containing the times and values of the solution, as well as various diagnostic messages.

Return type `object`

1.7.4 Scikits.odes Solvers

class `pybamm.ScikitsOdeSolver` (*method*='cvmde', *tol*=1e-08, *linsolver*='dense')

Solve a discretised model, using `scikits.odes`.

Parameters

- **method** (*str*, *optional*) – The method to use in `solve_ivp` (default is “BDF”)
- **tolerance** (*float*, *optional*) – The tolerance for the solver (default is 1e-8). Set as the both `reltol` and `abstol` in `solve_ivp`.
- **linsolver** (*str*, *optional*) – Can be ‘dense’ (= default), ‘lapackdense’, ‘spgmr’, ‘spbcgs’, ‘sptfqmr’

integrate (*derivs*, *y0*, *t_eval*, *events*=None, *mass_matrix*=None, *jacobian*=None)

Solve a model defined by `dydt` with initial conditions `y0`.

Parameters

- **derivs** (*method*) – A function that takes in `t` and `y` and returns the time-derivative `dydt`
- **y0** (*numeric type*) – The initial conditions
- **t_eval** (*numeric type*) – The times at which to compute the solution

- **events** (*method, optional*) – A function that takes in *t* and *y* and returns conditions for the solver to stop
- **mass_matrix** (*array_like, optional*) – The (sparse) mass matrix for the chosen spatial method.
- **jacobian** (*method, optional*) – A function that takes in *t* and *y* and returns the Jacobian. If *None*, the solver will approximate the Jacobian. (see *SUNDIALS docs*. <<https://computation.llnl.gov/projects/sundials>>).

```
class pybamm.ScikitsDaeSolver(method='ida', tol=1e-08, root_method='lm', root_tol=1e-06,  
                             max_steps=1000)
```

Solve a discretised model, using scikits.odes.

Parameters

- **method** (*str, optional*) – The method to use in `solve_ivp` (default is “BDF”)
- **tolerance** (*float, optional*) – The tolerance for the solver (default is 1e-8). Set as the both `reltol` and `abstol` in `solve_ivp`.
- **root_method** (*str, optional*) – The method to use to find initial conditions (default is “lm”)
- **tolerance** – The tolerance for the initial-condition solver (default is 1e-8).
- **max_steps** (*int, optional*) – The maximum number of steps the solver will take before terminating (default is 1000).

```
integrate (residuals, y0, t_eval, events=None, mass_matrix=None, jacobian=None)
```

Solve a DAE model defined by residuals with initial conditions *y0*.

Parameters

- **residuals** (*method*) – A function that takes in *t*, *y* and *ydots* and returns the residuals of the equations
- **y0** (*numeric type*) – The initial conditions
- **t_eval** (*numeric type*) – The times at which to compute the solution
- **events** (*method, optional*) – A function that takes in *t* and *y* and returns conditions for the solver to stop
- **mass_matrix** (*array_like, optional*) – The (sparse) mass matrix for the chosen spatial method.
- **jacobian** (*method, optional*) – A function that takes in *t* and *y* and returns the Jacobian. If *None*, the solver will approximate the Jacobian. (see *SUNDIALS docs*. <<https://computation.llnl.gov/projects/sundials>>).

1.7.5 Solution

```
class pybamm.Solution(t, y, t_event, y_event, termination)
```

Class containing the solution of, and various attributes associated with, a PyBaMM model.

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

append (`solution`)

Appends `solution.t` and `solution.y` onto `self.t` and `self.y`. Note: 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]`).

t

Times at which the solution is evaluated

t_event

Time at which the event happens

termination

Reason for termination

y

Values of the solution

y_event

Value of the solution at the time of the event

1.8 Post-Process Variables

`pybamm.post_process_variables` (`variables`, `t_sol`, `u_sol`, `mesh=None`, `interp_kind='linear'`)

Post-process all variables in a model

Parameters

- **variables** (`dict`) – Dictionary of variables
- **t_sol** (`array_like`, size (m,)) – The time vector returned by the solver
- **u_sol** (`array_like`, size (m, k)) – The solution vector returned by the solver. Can include solution values that other than those that get read by `base_variable.evaluate()` (i.e. $k \geq n$)
- **mesh** (`pybamm.Mesh`) – The mesh used to solve, used here to calculate the reference x values for interpolation
- **interp_kind** (`str`) – The method to use for interpolation

Returns Dictionary of processed variables

Return type `dict`

class `pybamm.ProcessedVariable` (`base_variable`, `t_sol`, `u_sol`, `mesh=None`, `interp_kind='linear'`, `known_evals=None`)

An object that can be evaluated at arbitrary (scalars or vectors) `t` and `x`, and returns the (interpolated) value of the base variable at that `t` and `x`.

Parameters

- **base_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)

- **t_sol**(*array_like*, *size* (*m*,)) – The time vector returned by the solver
- **u_sol**(*array_like*, *size* (*m*, *k*)) – The solution vector returned by the solver. Can include solution values that other than those that get read by `base_variable.evaluate()` (i.e. $k \geq n$)
- **mesh**(*pybamm.Mesh*) – The mesh used to solve, used here to calculate the reference x values for interpolation
- **interp_kind**(*str*) – The method to use for interpolation

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

call_3D(*t*, *x*, *r*, *y*, *z*)
Evaluate a 3D variable

initialise_3D()
Initialise a 3D object that depends on x and r. Needs to be generalised to deal with other domains

1.9 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.

CHAPTER 2

Examples

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

There are many ways to contribute to PyBaMM:

3.1 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!

3.1.1 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](#).

3.1.2 Adding a set of parameters values

Parameter sets should be added as csv files in the appropriate chemistry folder in `input/parameters/` (add a new folder if no parameters exist for that chemistry yet). The expected structure of the csv file is

| Name | Value | Units | Reference | Notes |
|---------|-------|-------------------|------------|------------|
| Example | 13 | m.s ⁻² | bloggs2019 | an example |

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

3.1.3 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/lead-acid/electrolyte_diffusivity_Bloggs2019.py`

```
def electrolyte_diffusivity_Bloggs2019(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
```

3.1.4 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.

3.1.5 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()  
    input_path = os.path.join(os.getcwd(), "path", "to", "functions")  
    parameter_values = pybamm.ParameterValues(  
        "path/to/parameter/file.csv",  
        {  
            "Typical current [A]": 1,  
            "Current function": os.path.join(  
                os.getcwd(),  
                "pybamm",  
                "parameters",  
                "standard_current_functions",  
                "constant_current.py",  
            ),  
            "First function": os.path.join(input_path, "first_function.py"),  
            "Second function": os.path.join(input_path, "second_function.py"),  
        },  
    )  
  
    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.

3.2 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.

3.2.1 The role of models

One of the main motivations for PyBaMM is to allow for new models of batteries to be easily 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.

3.2.2 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 is 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 quantity 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 an instance of `pybamm.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.

3.2.3 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.

3.3 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!

3.3.1 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 \rightarrow Addition), and calls upon spatial methods to perform the harder conversions (e.g. `grad(u)` \rightarrow Matrix * StateVector, SpatialVariable \rightarrow Vector, etc).

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

3.3.2 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](#).

3.3.3 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) $\text{grad}(x^2) = 2*x$, $\text{integral}(\sin(x), 0, \pi) = 2$
- (more advanced) make sure that the operations converge at the correct rate to known analytical solutions as you decrease the grid size

3.3.4 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.

3.4 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!

3.4.1 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 .

3.4.2 Base solver classes vs specific solver classes

There is one general base solver class, *pybamm.BaseSolver*, and two specialised base classes, *pybamm.OdeSolver* and *pybamm.DaeSolver*. The general base class simply sets up some useful solver properties such as tolerances. The specialised base classes 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 DAE solver class also computes a consistent set of initial conditions for the algebraic equations, using `model.concatenated_initial_conditions` as an initial guess.

3.4.3 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 either `pybamm.OdeSolver` or `pybamm.DaeSolver`, depending on whether the new solver can solve DAE systems. For example:

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

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 (the interfaces are slightly different for an ODE Solver and a DAE Solver, see `pybamm.OdeSolver.integrate()` vs `pybamm.DaeSolver.integrate()`)

For an example of an existing solver implementation, see the Scikits DAE solver [API docs](#) and [notebook](#).

3.4.4 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/`.

3.4.5 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](#).

p

- pybamm, [3](#)
- pybamm.parameters.electrical_parameters,
[67](#)
- pybamm.parameters.geometric_parameters,
[67](#)
- pybamm.parameters.standard_parameters_lead_acid,
[67](#)
- pybamm.parameters.standard_parameters_lithium_ion,
[67](#)
- pybamm.parameters.thermal_parameters,
[67](#)

Symbols

`__abs__()` (*pybamm.Symbol method*), 3
`__add__()` (*pybamm.Symbol method*), 3
`__getitem__()` (*pybamm.Symbol method*), 3
`__init__()` (*pybamm.Symbol method*), 3
`__matmul__()` (*pybamm.Symbol method*), 3
`__mul__()` (*pybamm.Symbol method*), 3
`__neg__()` (*pybamm.Symbol method*), 3
`__pow__()` (*pybamm.Symbol method*), 4
`__radd__()` (*pybamm.Symbol method*), 4
`__repr__()` (*pybamm.Symbol method*), 4
`__rmatmul__()` (*pybamm.Symbol method*), 4
`__rmul__()` (*pybamm.Symbol method*), 4
`__rpow__()` (*pybamm.Symbol method*), 4
`__rsub__()` (*pybamm.Symbol method*), 4
`__rtruediv__()` (*pybamm.Symbol method*), 4
`__str__()` (*pybamm.Symbol method*), 4
`__sub__()` (*pybamm.Symbol method*), 4
`__truediv__()` (*pybamm.Symbol method*), 4

A

AbsoluteValue (*class in pybamm*), 13
 add_domain() (*pybamm.Geometry method*), 68
 add_ghost_meshes() (*pybamm.Mesh method*), 70
 add_ghost_nodes() (*pybamm.FiniteVolume method*), 77
 Addition (*class in pybamm*), 11
 algebraic (*pybamm.BaseModel attribute*), 23
 algebraic (*pybamm.BaseSubModel attribute*), 29
 AlgebraicSolver (*class in pybamm*), 84
 append() (*pybamm.Solution method*), 90
 assemble_mass_form() (*pybamm.ScikitFiniteElement method*), 81

B

BackwardTafel (*class in pybamm.interface.kinetics*), 51
 BaseBatteryModel (*class in pybamm*), 25

BaseCompositePotentialPair (*class in pybamm.current_collector*), 31
 BaseElectrode (*class in pybamm.electrode*), 36
 BaseElectrolyteConductivity (*class in pybamm.electrolyte*), 39
 BaseElectrolyteDiffusion (*class in pybamm.electrolyte*), 39
 BaseFirstOrderKinetics (*class in pybamm.interface.kinetics*), 50
 BaseHigherOrderModel (*class in pybamm.lead_acid*), 27
 BaseInterface (*class in pybamm.interface*), 47
 BaseInverseFirstOrderKinetics (*class in pybamm.interface.inverse_kinetics*), 48
 BaseInverseKinetics (*class in pybamm.interface.inverse_kinetics*), 48
 BaseModel (*class in pybamm*), 23
 BaseModel (*class in pybamm.convection*), 34
 BaseModel (*class in pybamm.current_collector*), 30
 BaseModel (*class in pybamm.electrode.ohm*), 36
 BaseModel (*class in pybamm.electrolyte.stefan_maxwell.conductivity*), 40
 BaseModel (*class in pybamm.electrolyte.stefan_maxwell.diffusion*), 44
 BaseModel (*class in pybamm.interface.diffusion_limited*), 47
 BaseModel (*class in pybamm.interface.kinetics*), 49
 BaseModel (*class in pybamm.lead_acid*), 27
 BaseModel (*class in pybamm.lithium_ion*), 26
 BaseModel (*class in pybamm.oxygen_diffusion*), 52
 BaseModel (*class in pybamm.particle.fast*), 57
 BaseModel (*class in pybamm.particle.fickian*), 55
 BaseModel (*class in pybamm.porosity*), 58
 BaseModel (*class in pybamm.thermal.x_full*), 61
 BaseModel (*class in pybamm.thermal.x_lumped*), 62
 BaseModel (*class in pybamm.thermal.xyz_lumped*), 64
 BaseParticle (*class in pybamm.particle*), 55
 BasePotentialPair (*class in py-*

- bamm.current_collector*), 32
- BaseQuiteConductivePotentialPair (class in *pybamm.current_collector*), 33
- BaseSolver (class in *pybamm*), 85
- BaseSubModel (class in *pybamm*), 29
- BaseThermal (class in *pybamm.thermal*), 65
- bc_apply() (*pybamm.ScikitFiniteElement* method), 81
- BinaryOperator (class in *pybamm*), 10
- boundary_conditions (*pybamm.BaseModel* attribute), 23
- boundary_conditions (*pybamm.BaseSubModel* attribute), 29
- boundary_integral() (*pybamm.ScikitFiniteElement* method), 81
- boundary_integral() (*pybamm.SpatialMethod* method), 74
- boundary_integral_vector() (*pybamm.ScikitFiniteElement* method), 81
- boundary_mass_matrix() (*pybamm.ScikitFiniteElement* method), 82
- boundary_value() (in module *pybamm*), 18
- boundary_value_or_flux() (*bamm.FiniteVolume* method), 78
- boundary_value_or_flux() (*pybamm.ScikitFiniteElement* method), 82
- boundary_value_or_flux() (*pybamm.SpatialMethod* method), 74
- BoundaryGradient (class in *pybamm*), 17
- BoundaryIntegral (class in *pybamm*), 15
- BoundaryOperator (class in *pybamm*), 16
- BoundaryValue (class in *pybamm*), 16
- Broadcast (class in *pybamm*), 19
- broadcast() (*pybamm.SpatialMethod* method), 74
- ButlerVolmer (class in *pybamm.interface.kinetics*), 50
- ButlerVolmer (class in *pybamm.interface.lead_acid*), 51
- ButlerVolmer (class in *pybamm.interface.lithium_ion*), 51
- C**
- calculate_consistent_initial_conditions (*pybamm.DaeSolver* method), 87
- call_2D() (*pybamm.ProcessedVariable* method), 91
- call_3D() (*pybamm.ProcessedVariable* method), 91
- check_algebraic_equations() (*pybamm.BaseModel* method), 24
- check_and_set_domain_and_broadcast_type (*pybamm.Broadcast* method), 19
- check_default_variables_dictionaries() (*pybamm.BaseModel* method), 24
- check_ics_bcs() (*pybamm.BaseModel* method), 24
- check_initial_conditions() (*pybamm.Discretisation* method), 72
- check_initial_conditions_rhs() (*pybamm.Discretisation* method), 72
- check_model() (*pybamm.Discretisation* method), 72
- check_tab_conditions() (*pybamm.Discretisation* method), 72
- check_variables() (*pybamm.Discretisation* method), 72
- check_well_determined() (*pybamm.BaseModel* method), 24
- check_well_posedness() (*pybamm.BaseModel* method), 24
- children (*pybamm.Symbol* attribute), 4
- combine_submeshes() (*pybamm.Mesh* method), 70
- Composite (class in *pybamm.convection*), 35
- Composite (class in *pybamm.electrode.ohm*), 37
- Composite (class in *pybamm.electrolyte.stefan_maxwell.conductivity*), 41
- Composite (class in *pybamm.electrolyte.stefan_maxwell.diffusion*), 45
- Composite (class in *pybamm.lead_acid*), 28
- Composite (class in *pybamm.oxygen_diffusion*), 52
- CompositeExtended (class in *pybamm.lead_acid*), 28
- CompositePotentialPair1plus1D (class in *pybamm.current_collector*), 31
- CompositePotentialPair2plus1D (class in *pybamm.current_collector*), 31
- compute_solution() (*pybamm.BaseSolver* method), 85
- compute_solution() (*pybamm.DaeSolver* method), 87
- compute_solution() (*pybamm.OdeSolver* method), 86
- concatenated_algebraic (*pybamm.BaseModel* attribute), 24
- concatenated_initial_conditions (*pybamm.BaseModel* attribute), 24
- concatenated_rhs (*pybamm.BaseModel* attribute), 23
- Concatenation (class in *pybamm*), 18
- concatenation() (*pybamm.FiniteVolume* method), 78
- concatenation() (*pybamm.SpatialMethod* method), 75
- Constant (class in *pybamm.porosity*), 59
- ConstantConcentration (class in *pybamm.electrolyte.stefan_maxwell.diffusion*), 44
- Cos (class in *pybamm*), 22
- cos() (in module *pybamm*), 22
- Cosh (class in *pybamm*), 22
- cosh() (in module *pybamm*), 22

| | |
|--|---|
| <code>create_mass_matrix()</code> (<i>pybamm.Discretisation method</i>), 72 | <code>delta_function()</code> (<i>pybamm.SpatialMethod method</i>), 75 |
| <code>CurrentCollector0D</code> (<i>class in pybamm.thermal.x_lumped</i>), 63 | <code>DeltaFunction</code> (<i>class in pybamm</i>), 16 |
| <code>CurrentCollector1D</code> (<i>class in pybamm.thermal.x_lumped</i>), 63 | <code>DFN</code> (<i>class in pybamm.lithium_ion</i>), 27 |
| <code>CurrentCollector1D</code> (<i>class in pybamm.thermal.xyz_lumped</i>), 65 | <code>diff()</code> (<i>pybamm.AbsoluteValue method</i>), 13 |
| <code>CurrentCollector2D</code> (<i>class in pybamm.thermal.x_lumped</i>), 64 | <code>diff()</code> (<i>pybamm.Function method</i>), 21 |
| <code>CurrentCollector2D</code> (<i>class in pybamm.thermal.xyz_lumped</i>), 65 | <code>diff()</code> (<i>pybamm.FunctionParameter method</i>), 7 |
| | <code>diff()</code> (<i>pybamm.Kron method</i>), 12 |
| | <code>diff()</code> (<i>pybamm.MatrixMultiplication method</i>), 11 |
| | <code>diff()</code> (<i>pybamm.Outer method</i>), 12 |
| | <code>diff()</code> (<i>pybamm.SpatialOperator method</i>), 14 |
| | <code>diff()</code> (<i>pybamm.Symbol method</i>), 4 |
| | <code>Discretisation</code> (<i>class in pybamm</i>), 71 |
| | <code>div()</code> (<i>in module pybamm</i>), 17 |
| | <code>Divergence</code> (<i>class in pybamm</i>), 14 |
| | <code>divergence()</code> (<i>pybamm.FiniteVolume method</i>), 78 |
| | <code>divergence()</code> (<i>pybamm.ScikitFiniteElement method</i>), 82 |
| | <code>divergence()</code> (<i>pybamm.SpatialMethod method</i>), 75 |
| | <code>divergence_matrix()</code> (<i>pybamm.FiniteVolume method</i>), 78 |
| | <code>Division</code> (<i>class in pybamm</i>), 11 |
| | <code>domain</code> (<i>pybamm.Symbol attribute</i>), 4 |
| | <code>DomainConcatenation</code> (<i>class in pybamm</i>), 18 |
| D | |
| <code>DaeSolver</code> (<i>class in pybamm</i>), 86 | |
| <code>default_solver</code> (<i>pybamm.BaseBatteryModel attribute</i>), 25 | |
| <code>default_solver</code> (<i>pybamm.electrode.BaseElectrode attribute</i>), 36 | |
| <code>default_solver</code> (<i>pybamm.electrode.ohm.BaseModel attribute</i>), 36 | |
| <code>default_solver</code> (<i>pybamm.electrode.ohm.Composite attribute</i>), 37 | |
| <code>default_solver</code> (<i>pybamm.electrode.ohm.Full attribute</i>), 38 | |
| <code>default_solver</code> (<i>pybamm.electrode.ohm.LeadingOrder attribute</i>), 37 | |
| <code>default_solver</code> (<i>pybamm.electrode.ohm.SurfaceForm attribute</i>), 39 | |
| <code>default_solver</code> (<i>pybamm.electrolyte.stefan_maxwell.conductivity.Full attribute</i>), 41 | |
| <code>default_solver</code> (<i>pybamm.lead_acid.BaseHigherOrderModel attribute</i>), 28 | |
| <code>default_solver</code> (<i>pybamm.lead_acid.Full attribute</i>), 28 | |
| <code>default_solver</code> (<i>pybamm.lead_acid.LOQS attribute</i>), 27 | |
| <code>default_solver</code> (<i>pybamm.lithium_ion.DFN attribute</i>), 27 | |
| <code>default_solver</code> (<i>pybamm.lithium_ion.SPM attribute</i>), 26 | |
| <code>default_solver</code> (<i>pybamm.lithium_ion.SPMe attribute</i>), 26 | |
| <code>definite_integral_matrix()</code> (<i>pybamm.FiniteVolume method</i>), 78 | |
| <code>definite_integral_matrix()</code> (<i>pybamm.ScikitFiniteElement method</i>), 82 | |
| <code>DefiniteIntegralVector</code> (<i>class in pybamm</i>), 15 | |
| <code>delta_function()</code> (<i>pybamm.FiniteVolume method</i>), 78 | |
| | <code>edge_to_node()</code> (<i>pybamm.FiniteVolume method</i>), 79 |
| | <code>EffectiveResistance2D</code> (<i>class in pybamm.current_collector</i>), 31 |
| | <code>evaluate()</code> (<i>pybamm.BinaryOperator method</i>), 11 |
| | <code>evaluate()</code> (<i>pybamm.Concatenation method</i>), 18 |
| | <code>evaluate()</code> (<i>pybamm.EvaluatorPython method</i>), 22 |
| | <code>evaluate()</code> (<i>pybamm.Function method</i>), 21 |
| | <code>evaluate()</code> (<i>pybamm.Symbol method</i>), 4 |
| | <code>evaluate()</code> (<i>pybamm.UnaryOperator method</i>), 13 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.BinaryOperator method</i>), 11 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.BoundaryIntegral method</i>), 16 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.BoundaryOperator method</i>), 16 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.Broadcast method</i>), 19 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.Concatenation method</i>), 18 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.DefiniteIntegralVector method</i>), 15 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.FullBroadcast method</i>), 19 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.Function method</i>), 21 |
| | <code>evaluate_for_shape()</code> (<i>pybamm.FunctionParameter method</i>), 7 |
| E | |

`evaluate_for_shape()` (*pybamm*.*IndefiniteIntegral* method), 15
`evaluate_for_shape()` (*pybamm*.*IndependentVariable* method), 8
`evaluate_for_shape()` (*pybamm*.*Index* method), 13
`evaluate_for_shape()` (*pybamm*.*Integral* method), 15
`evaluate_for_shape()` (*pybamm*.*Mass* method), 14
`evaluate_for_shape()` (*pybamm*.*Parameter* method), 7
`evaluate_for_shape()` (*pybamm*.*PrimaryBroadcast* method), 20
`evaluate_for_shape()` (*pybamm*.*StateVector* method), 10
`evaluate_for_shape()` (*pybamm*.*Symbol* method), 5
`evaluate_for_shape()` (*pybamm*.*Time* method), 8
`evaluate_for_shape()` (*pybamm*.*UnaryOperator* method), 13
`evaluate_for_shape()` (*pybamm*.*Variable* method), 8
`evaluate_ignoring_errors()` (*pybamm*.*Symbol* method), 5
`evaluates_on_edges()` (*pybamm*.*BinaryOperator* method), 11
`evaluates_on_edges()` (*pybamm*.*BoundaryIntegral* method), 16
`evaluates_on_edges()` (*pybamm*.*DeltaFunction* method), 16
`evaluates_on_edges()` (*pybamm*.*Divergence* method), 14
`evaluates_on_edges()` (*pybamm*.*Gradient* method), 14
`evaluates_on_edges()` (*pybamm*.*Gradient_Squared* method), 14
`evaluates_on_edges()` (*pybamm*.*Index* method), 13
`evaluates_on_edges()` (*pybamm*.*Inner* method), 12
`evaluates_on_edges()` (*pybamm*.*Integral* method), 15
`evaluates_on_edges()` (*pybamm*.*Laplacian* method), 14
`evaluates_on_edges()` (*pybamm*.*Symbol* method), 5
`evaluates_on_edges()` (*pybamm*.*UnaryOperator* method), 13
`evaluates_to_number()` (*pybamm*.*Symbol* method), 5
`evaluation_array` (*pybamm*.*StateVector* attribute), 10
`EvaluatorPython` (class in *pybamm*), 22

`events` (*pybamm*.*BaseModel* attribute), 23
`events` (*pybamm*.*BaseSubModel* attribute), 29
`exp()` (in module *pybamm*), 22
`Exponential` (class in *pybamm*), 22

F

`FiniteVolume` (class in *pybamm*), 77
`FirstOrder` (class in *pybamm*.*oxygen_diffusion*), 52
`FirstOrderButlerVolmer` (class in *pybamm*.*interface.kinetics*), 50
`FirstOrderButlerVolmer` (class in *pybamm*.*interface.lead_acid*), 51
`FirstOrderForwardTafel` (class in *pybamm*.*interface.kinetics*), 51
`FOQS` (class in *pybamm*.*lead_acid*), 28
`format()` (*pybamm*.*Timer* method), 92
`ForwardTafel` (class in *pybamm*.*interface.kinetics*), 51
`Full` (class in *pybamm*.*convection*), 35
`Full` (class in *pybamm*.*electrode.ohm*), 38
`Full` (class in *pybamm*.*electrolyte.stefan_maxwell.conductivity*), 41
`Full` (class in *pybamm*.*electrolyte.stefan_maxwell.diffusion*), 45
`Full` (class in *pybamm*.*lead_acid*), 28
`Full` (class in *pybamm*.*oxygen_diffusion*), 53
`Full` (class in *pybamm*.*porosity*), 60
`FullAlgebraic` (class in *pybamm*.*electrolyte.stefan_maxwell.conductivity.surface_potential*), 43
`FullBroadcast` (class in *pybamm*), 19
`FullDifferential` (class in *pybamm*.*electrolyte.stefan_maxwell.conductivity.surface_potential*), 42
`FullDiffusionLimited` (class in *pybamm*.*interface.diffusion_limited*), 47
`Function` (class in *pybamm*), 21
`FunctionParameter` (class in *pybamm*), 7

G

`Geometry` (class in *pybamm*), 68
`Geometry1DMacro` (class in *pybamm*), 69
`Geometry1DMicro` (class in *pybamm*), 69
`Geometry1pl1DMicro` (class in *pybamm*), 69
`Geometry2DCurrentCollector` (class in *pybamm*), 70
`Geometry3DMacro` (class in *pybamm*), 69
`Geometryxp0pl1DMicro` (class in *pybamm*), 69
`Geometryxpl1DMacro` (class in *pybamm*), 69
`Geometryxplpl1DMicro` (class in *pybamm*), 70
`get_children_auxiliary_domains()` (*pybamm*.*BinaryOperator* method), 11
`get_children_auxiliary_domains()` (*pybamm*.*Concatenation* method), 18

| | | | |
|--------------------------------------|---|---|---|
| <code>get_children_domains()</code> | (py- <i>bamm.BinaryOperator</i> method), 11 | <i>bamm.interface.inverse_kinetics.BaseInverseKinetics</i> method), 49 | |
| <code>get_children_domains()</code> | (pybamm. <i>Function</i> method), 21 | <code>get_coupled_variables()</code> | (py- <i>bamm.interface.kinetics.BaseFirstOrderKinetics</i> method), 50 |
| <code>get_children_domains()</code> | (py- <i>bamm.FunctionParameter</i> method), 7 | <code>get_coupled_variables()</code> | (py- <i>bamm.interface.kinetics.BaseModel</i> method), 49 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.BaseSubModel</i> method), 29 | <code>get_coupled_variables()</code> | (py- <i>bamm.oxygen_diffusion.Composite</i> method), 52 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.convection.Composite</i> method), 35 | <code>get_coupled_variables()</code> | (py- <i>bamm.oxygen_diffusion.FirstOrder</i> method), 53 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.convection.Full</i> method), 35 | <code>get_coupled_variables()</code> | (py- <i>bamm.oxygen_diffusion.Full</i> method), 53 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.convection.LeadinOrder</i> method), 34 | <code>get_coupled_variables()</code> | (py- <i>bamm.oxygen_diffusion.LeadinOrder</i> method), 54 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.current_collector.BaseModel</i> method), 30 | <code>get_coupled_variables()</code> | (py- <i>bamm.particle.fickian.ManyParticles</i> method), 56 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.current_collector.SingleParticlePotentialPair</i> method), 33 | <code>get_coupled_variables()</code> | (py- <i>bamm.particle.fickian.SingleParticle</i> method), 57 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrode.ohm.Composite</i> method), 37 | <code>get_coupled_variables()</code> | (py- <i>bamm.porosity.Full</i> method), 60 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrode.ohm.Full</i> method), 38 | <code>get_coupled_variables()</code> | (py- <i>bamm.porosity.LeadinOrder</i> method), 59 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrode.ohm.LeadinOrder</i> method), 37 | <code>get_coupled_variables()</code> | (py- <i>bamm.thermal.isothermal.Isothermal</i> method), 61 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrode.ohm.SurfaceForm</i> method), 39 | <code>get_coupled_variables()</code> | (py- <i>bamm.thermal.x_full.BaseModel</i> method), 61 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.conductivity.Full</i> method), 41 | <code>get_coupled_variables()</code> | (py- <i>bamm.thermal.x_lumped.BaseModel</i> method), 62 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.conductivity.LeadinOrder</i> method), 40 | <code>get_coupled_variables()</code> | (py- <i>bamm.thermal.xyz_lumped.BaseModel</i> method), 64 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.Composite</i> method), 45 | <code>get_fundamental_variables()</code> | (py- <i>bamm.BaseSubModel</i> method), 29 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.Full</i> method), 45 | <code>get_fundamental_variables()</code> | (py- <i>bamm.convection.Full</i> method), 35 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.LeadinOrder</i> method), 46 | <code>get_fundamental_variables()</code> | (py- <i>bamm.convection.NoConvection</i> method), 34 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.interface.diffusion_limited.BaseModel</i> method), 47 | <code>get_fundamental_variables()</code> | (py- <i>bamm.current_collector.BaseCompositePotentialPair</i> method), 31 |
| <code>get_coupled_variables()</code> | (py- <i>bamm.interface.inverse_kinetics.BaseInverseFirstOrderKinetics</i> method), 48 | <code>get_fundamental_variables()</code> | (py- <i>bamm.current_collector.BasePotentialPair</i> method), 32 |
| <code>get_coupled_variables()</code> | (py- | | |

| | | |
|--|---|--|
| <code>get_fundamental_variables()</code> | (py- <i>bamm.current_collector.BaseQuiteConductivePotentialPair</i> method), 33 | 62 <code>get_fundamental_variables()</code> (py- <i>bamm.thermal.xyz_lumped.BaseModel</i> method), 64 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.current_collector.Uniform</i> method), 31 | <code>get_infinite_nested_dict()</code> (in module <i>py- bamm</i>), 91 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.electrode.ohm.Full</i> method), 38 | <code>get_processed_potentials()</code> (py- <i>bamm.current_collector.EffectiveResistance2D</i> method), 31 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.conductivity.Full</i> method), 41 | <code>get_termination_reason()</code> (py- <i>bamm.BaseSolver</i> method), 85 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.ConstantConcentration</i> method), 44 | <code>grad()</code> (in module <i>pybamm</i>), 17 <code>get_concentration()</code> (in module <i>pybamm</i>), 17 <i>Gradient</i> (class in <i>pybamm</i>), 14 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.Full</i> method), 45 | <code>gradient()</code> (<i>pybamm.FiniteVolume</i> method), 79 <code>gradient()</code> (<i>pybamm.ScikitFiniteElement</i> method), 82 <code>gradient()</code> (<i>pybamm.SpatialMethod</i> method), 75 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.LeadingOrder</i> method), 46 | <code>gradient_matrix()</code> (<i>pybamm.FiniteVolume</i> method), 79 <i>Gradient_Squared</i> (class in <i>pybamm</i>), 14 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.oxygen_diffusion.Full</i> method), 53 | <code>gradient_squared()</code> (<i>pybamm.ScikitFiniteElement</i> method), 82 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.oxygen_diffusion.LeadingOrder</i> method), 54 | <code>gradient_squared()</code> (<i>pybamm.SpatialMethod</i> method), 75 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.oxygen_diffusion.NoOxygen</i> method), 55 | H <code>has_symbol_of_classes()</code> (<i>pybamm.Symbol</i> method), 5 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.particle.fast.ManyParticles</i> method), 58 | I <code>indefinite_integral()</code> (<i>pybamm.FiniteVolume</i> method), 79 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.particle.fast.SingleParticle</i> method), 58 | <code>indefinite_integral()</code> (py- <i>bamm.ScikitFiniteElement</i> method), 82 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.particle.fickian.ManyParticles</i> method), 56 | <code>indefinite_integral()</code> (<i>pybamm.SpatialMethod</i> method), 76 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.particle.fickian.SingleParticle</i> method), 57 | <code>indefinite_integral_matrix_edges()</code> (<i>pybamm.FiniteVolume</i> method), 79 <code>indefinite_integral_matrix_nodes()</code> (<i>pybamm.FiniteVolume</i> method), 79 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.porosity.Constant</i> method), 59 | <i>IndefiniteIntegral</i> (class in <i>pybamm</i>), 15 <i>IndependentVariable</i> (class in <i>pybamm</i>), 8 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.porosity.Full</i> method), 60 | <i>Index</i> (class in <i>pybamm</i>), 13 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.porosity.LeadingOrder</i> method), 59 | <code>initial_conditions</code> (<i>pybamm.BaseModel</i> at- tribute), 23 <code>initial_conditions</code> (<i>pybamm.BaseSubModel</i> at- tribute), 29 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.thermal.isothermal.Isothermal</i> method), 61 | <code>initialise_3D()</code> (<i>pybamm.ProcessedVariable</i> method), 91 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.thermal.x_full.BaseModel</i> method), 61 | <i>Inner</i> (class in <i>pybamm</i>), 11 <i>Integral</i> (class in <i>pybamm</i>), 14 |
| <code>get_fundamental_variables()</code> | (py- <i>bamm.thermal.x_lumped.BaseModel</i> method), | <code>integral()</code> (<i>pybamm.FiniteVolume</i> method), 79 <code>integral()</code> (<i>pybamm.ScikitFiniteElement</i> method), 83 <code>integral()</code> (<i>pybamm.SpatialMethod</i> method), 76 |

`integrate()` (*pybamm.DaeSolver method*), 87
`integrate()` (*pybamm.OdeSolver method*), 86
`integrate()` (*pybamm.ScikitsDaeSolver method*), 89
`integrate()` (*pybamm.ScikitsOdeSolver method*), 88
`integrate()` (*pybamm.ScipySolver method*), 88
`internal_neumann_condition()` (*pybamm.FiniteVolume method*), 80
`internal_neumann_condition()` (*pybamm.SpatialMethod method*), 76
`InverseButlerVolmer` (class in *pybamm.interface.inverse_kinetics*), 49
`InverseButlerVolmer` (class in *pybamm.interface.lead_acid*), 51
`InverseButlerVolmer` (class in *pybamm.interface.lithium_ion*), 51
`InverseFirstOrderKinetics` (class in *pybamm.interface.lead_acid*), 51
`is_constant()` (*pybamm.Symbol method*), 5
`Isothermal` (class in *pybamm.thermal.isothermal*), 60

J

`jac()` (*pybamm.AbsoluteValue method*), 13
`jac()` (*pybamm.Kron method*), 12
`jac()` (*pybamm.Scalar method*), 9
`jac()` (*pybamm.SpatialOperator method*), 14
`jac()` (*pybamm.StateVector method*), 10
`jac()` (*pybamm.Symbol method*), 5
`jacobian` (*pybamm.BaseModel attribute*), 24

K

`Kron` (class in *pybamm*), 12

L

`Laplacian` (class in *pybamm*), 14
`laplacian()` (in module *pybamm*), 17
`laplacian()` (*pybamm.FiniteVolume method*), 80
`laplacian()` (*pybamm.ScikitFiniteElement method*), 83
`laplacian()` (*pybamm.SpatialMethod method*), 76
`LeadingOrder` (class in *pybamm.convection*), 34
`LeadingOrder` (class in *pybamm.electrode.ohm*), 37
`LeadingOrder` (class in *pybamm.electrolyte.stefan_maxwell.conductivity*), 40
`LeadingOrder` (class in *pybamm.electrolyte.stefan_maxwell.diffusion*), 46
`LeadingOrder` (class in *pybamm.oxygen_diffusion*), 54
`LeadingOrder` (class in *pybamm.porosity*), 59
`LeadingOrderAlgebraic` (class in *pybamm.electrolyte.stefan_maxwell.conductivity.surface_potential_form*), 43

`LeadingOrderDifferential` (class in *pybamm.electrolyte.stefan_maxwell.conductivity.surface_potential_form*), 43
`LeadingOrderDiffusionLimited` (class in *pybamm.interface.diffusion_limited*), 48
`load_function()` (in module *pybamm*), 91
`Log` (class in *pybamm*), 22
`log()` (in module *pybamm*), 22
`LOQS` (class in *pybamm.lead_acid*), 27

M

`ManyParticles` (class in *pybamm.particle.fast*), 58
`ManyParticles` (class in *pybamm.particle.fickian*), 56
`Mass` (class in *pybamm*), 14
`mass_matrix` (*pybamm.BaseModel attribute*), 24
`mass_matrix()` (*pybamm.ScikitFiniteElement method*), 83
`mass_matrix()` (*pybamm.SpatialMethod method*), 76
`mass_matrix()` (*pybamm.ZeroDimensionalMethod method*), 84
`Matrix` (class in *pybamm*), 9
`MatrixMultiplication` (class in *pybamm*), 11
`max()` (in module *pybamm*), 22
`Mesh` (class in *pybamm*), 70
`min()` (in module *pybamm*), 22
`Multiplication` (class in *pybamm*), 11

N

`name` (*pybamm.BaseModel attribute*), 23
`name` (*pybamm.Symbol attribute*), 5
`Negate` (class in *pybamm*), 13
`new_copy()` (*pybamm.BinaryOperator method*), 11
`new_copy()` (*pybamm.Concatenation method*), 18
`new_copy()` (*pybamm.Function method*), 21
`new_copy()` (*pybamm.FunctionParameter method*), 7
`new_copy()` (*pybamm.Parameter method*), 7
`new_copy()` (*pybamm.Scalar method*), 9
`new_copy()` (*pybamm.SpatialVariable method*), 8
`new_copy()` (*pybamm.StateVector method*), 10
`new_copy()` (*pybamm.Symbol method*), 5
`new_copy()` (*pybamm.Time method*), 8
`new_copy()` (*pybamm.UnaryOperator method*), 13
`new_copy()` (*pybamm.Variable method*), 8
`NoConvection` (class in *pybamm.convection*), 34
`NoCurrentCollector` (class in *pybamm.thermal.x_full*), 62
`NoCurrentCollector` (class in *pybamm.thermal.x_lumped*), 63
`node_to_edge()` (*pybamm.FiniteVolume method*), 80
`NoOxygen` (class in *pybamm.oxygen_diffusion*), 55
`NoReaction` (class in *pybamm.interface.kinetics*), 50
`NumpyConcatenation` (class in *pybamm*), 18

O

OdeSolver (class in pybamm), 86
 on_boundary() (pybamm.Scikit2DSubMesh method), 71
 options (pybamm.BaseBatteryModel attribute), 25
 options (pybamm.BaseModel attribute), 23
 orphans (pybamm.Symbol attribute), 6
 Outer (class in pybamm), 12
 outer() (in module pybamm), 12

P

param (pybamm.BaseSubModel attribute), 29
 Parameter (class in pybamm), 7
 ParameterValues (class in pybamm), 65
 post_process_variables() (in module pybamm), 90
 PotentialPair1plus1D (class in pybamm.current_collector), 32
 PotentialPair2plus1D (class in pybamm.current_collector), 32
 Power (class in pybamm), 11
 pre_order() (pybamm.Symbol method), 6
 PrimaryBroadcast (class in pybamm), 19
 print_evaluated_parameters() (in module pybamm), 67
 print_parameters() (in module pybamm), 67
 process_binary_operators() (pybamm.FiniteVolume method), 80
 process_binary_operators() (pybamm.SpatialMethod method), 77
 process_boundary_conditions() (pybamm.Discretisation method), 72
 process_dict() (pybamm.Discretisation method), 72
 process_geometry() (pybamm.ParameterValues method), 65
 process_initial_conditions() (pybamm.Discretisation method), 73
 process_model() (pybamm.Discretisation method), 73
 process_model() (pybamm.ParameterValues method), 66
 process_parameters_and_discretise() (pybamm.BaseBatteryModel method), 25
 process_rhs_and_algebraic() (pybamm.Discretisation method), 73
 process_symbol() (pybamm.Discretisation method), 73
 process_symbol() (pybamm.ParameterValues method), 66
 ProcessedVariable (class in pybamm), 90
 pybamm (module), 3
 pybamm.parameters.electrical_parameters (module), 67

pybamm.parameters.geometric_parameters (module), 67
 pybamm.parameters.standard_parameters_lead_acid (module), 67
 pybamm.parameters.standard_parameters_lithium_ion (module), 67
 pybamm.parameters.thermal_parameters (module), 67

Q

QuiteConductivePotentialPair1plus1D (class in pybamm.current_collector), 33
 QuiteConductivePotentialPair2plus1D (class in pybamm.current_collector), 33

R

read_parameters_csv() (pybamm.ParameterValues method), 66
 relabel_tree() (pybamm.Symbol method), 6
 render() (pybamm.Symbol method), 6
 reset() (pybamm.Timer method), 92
 rhs (pybamm.BaseModel attribute), 23
 rhs (pybamm.BaseSubModel attribute), 29
 rmse() (in module pybamm), 91
 root() (pybamm.AlgebraicSolver method), 84
 root_dir() (in module pybamm), 91

S

Scalar (class in pybamm), 9
 Scikit2DSubMesh (class in pybamm), 71
 ScikitFiniteElement (class in pybamm), 81
 ScikitsDaeSolver (class in pybamm), 89
 ScikitsOdeSolver (class in pybamm), 88
 ScipySolver (class in pybamm), 88
 set_algebraic() (pybamm.BaseSubModel method), 30
 set_algebraic() (pybamm.convection.Full method), 35
 set_algebraic() (pybamm.current_collector.BasePotentialPair method), 32
 set_algebraic() (pybamm.current_collector.BaseQuiteConductivePotentialPair method), 33
 set_algebraic() (pybamm.electrode.ohm.Full method), 38
 set_algebraic() (pybamm.electrolyte.stefan_maxwell.conductivity.Full method), 42
 set_algebraic() (pybamm.electrolyte.stefan_maxwell.conductivity.surface_potential method), 43

| | | | |
|--|---|---|---|
| <code>set_algebraic()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.conductivity.surface_potential_for_ion_exchange</i> method), 39 | <code>set_events()</code> | (pybamm. <i>particle.BaseParticle</i> method), 55 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.BaseSubModel</i> method), 30 | <code>set_full_convection_submodel()</code> | (py- <i>bamm.lead_acid.BaseHigherOrderModel</i> method), 28 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.convection.Full</i> method), 35 | <code>set_full_interface_submodel()</code> | (py- <i>bamm.lead_acid.BaseHigherOrderModel</i> method), 28 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.current_collector.PotentialPair1plus1D</i> method), 32 | <code>set_full_porosity_submodel()</code> | (py- <i>bamm.lead_acid.BaseHigherOrderModel</i> method), 28 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.current_collector.PotentialPair2plus1D</i> method), 32 | <code>set_full_porosity_submodel()</code> | (py- <i>bamm.lead_acid.Composite</i> method), 28 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.electrode.ohm.BaseModel</i> method), 36 | <code>set_full_porosity_submodel()</code> | (py- <i>bamm.lead_acid.CompositeExtended</i> method), 28 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.electrode.ohm.Composite</i> method), 37 | <code>set_full_porosity_submodel()</code> | (py- <i>bamm.lead_acid.FOQS</i> method), 28 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.electrode.ohm.Full</i> method), 38 | <code>set_id()</code> | (pybamm. <i>BoundaryIntegral</i> method), 16 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.electrode.ohm.LeadinOrder</i> method), 37 | <code>set_id()</code> | (pybamm. <i>BoundaryOperator</i> method), 16 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.conductivity.BaseModel</i> method), 40 | <code>set_id()</code> | (pybamm. <i>DefiniteIntegralVector</i> method), 15 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.conductivity.Full</i> method), 42 | <code>set_id()</code> | (pybamm. <i>DeltaFunction</i> method), 16 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.BaseModel</i> method), 44 | <code>set_id()</code> | (pybamm. <i>FunctionParameter</i> method), 7 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.ConstantConcentration</i> method), 44 | <code>set_id()</code> | (pybamm. <i>Index</i> method), 13 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.oxygen_diffusion.Full</i> method), 53 | <code>set_id()</code> | (pybamm. <i>Integral</i> method), 15 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.particle.fickian.BaseModel</i> method), 56 | <code>set_id()</code> | (pybamm. <i>Scalar</i> method), 9 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.thermal.x_full.NoCurrentCollector</i> method), 62 | <code>set_id()</code> | (pybamm. <i>StateVector</i> method), 10 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.thermal.x_lumped.CurrentCollector1D</i> method), 63 | <code>set_id()</code> | (pybamm. <i>Symbol</i> method), 6 |
| <code>set_boundary_conditions()</code> | (py- <i>bamm.thermal.x_lumped.CurrentCollector2D</i> method), 64 | <code>set_initial_conditions()</code> | (py- <i>bamm.BaseSubModel</i> method), 30 |
| <code>set_evaluation_array()</code> | (pybamm. <i>StateVector</i> method), 10 | <code>set_initial_conditions()</code> | (py- <i>bamm.convection.Full</i> method), 36 |
| <code>set_events()</code> | (pybamm. <i>BaseSubModel</i> method), 30 | <code>set_initial_conditions()</code> | (py- <i>bamm.current_collector.BasePotentialPair</i> method), 32 |
| <code>set_events()</code> | (pybamm. <i>electrolyte.BaseElectrolyteDiffusion</i> method), 46 | <code>set_initial_conditions()</code> | (py- <i>bamm.current_collector.BaseQuiteConductivePotentialPair</i> method), 33 |
| | | <code>set_initial_conditions()</code> | (py- <i>bamm.electrode.ohm.Full</i> method), 38 |
| | | <code>set_initial_conditions()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.conductivity.Full</i> method), 42 |
| | | <code>set_initial_conditions()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.Full</i> method), 46 |
| | | <code>set_initial_conditions()</code> | (py- <i>bamm.electrolyte.stefan_maxwell.diffusion.LeadinOrder</i> method), 46 |
| | | <code>set_initial_conditions()</code> | (py- <i>bamm.oxygen_diffusion.Full</i> method), 53 |

`set_initial_conditions()` (`pybamm.oxygen_diffusion.LeadingOrder method`), 54
`set_initial_conditions()` (`pybamm.particle.BaseParticle method`), 55
`set_initial_conditions()` (`pybamm.porosity.Full method`), 60
`set_initial_conditions()` (`pybamm.porosity.LeadingOrder method`), 59
`set_initial_conditions()` (`pybamm.thermal.x_full.BaseModel method`), 61
`set_initial_conditions()` (`pybamm.thermal.x_lumped.BaseModel method`), 63
`set_initial_conditions()` (`pybamm.thermal.xyz_lumped.BaseModel method`), 64
`set_internal_boundary_conditions()` (`pybamm.Discretisation method`), 73
`set_rhs()` (`pybamm.BaseSubModel method`), 30
`set_rhs()` (`pybamm.electrolyte.stefan_maxwell.conductivity.Full method`), 42
`set_rhs()` (`pybamm.electrolyte.stefan_maxwell.conductivity.surface_potential_form.FullDifferential method`), 42
`set_rhs()` (`pybamm.electrolyte.stefan_maxwell.conductivity.surface_potential_form.LeadOrderDifferential method`), 43
`set_rhs()` (`pybamm.electrolyte.stefan_maxwell.diffusion.Composite method`), 45
`set_rhs()` (`pybamm.electrolyte.stefan_maxwell.diffusion.Full method`), 46
`set_rhs()` (`pybamm.electrolyte.stefan_maxwell.diffusion.LeadOrder method`), 47
`set_rhs()` (`pybamm.oxygen_diffusion.Composite method`), 52
`set_rhs()` (`pybamm.oxygen_diffusion.Full method`), 54
`set_rhs()` (`pybamm.oxygen_diffusion.LeadingOrder method`), 54
`set_rhs()` (`pybamm.particle.fast.BaseModel method`), 57
`set_rhs()` (`pybamm.particle.fickian.BaseModel method`), 56
`set_rhs()` (`pybamm.porosity.Full method`), 60
`set_rhs()` (`pybamm.porosity.LeadingOrder method`), 59
`set_rhs()` (`pybamm.thermal.x_full.NoCurrentCollector method`), 62
`set_rhs()` (`pybamm.thermal.x_lumped.CurrentCollector0D method`), 63
`set_rhs()` (`pybamm.thermal.x_lumped.CurrentCollector1D method`), 63
`set_rhs()` (`pybamm.thermal.x_lumped.CurrentCollector2D method`), 64
`set_rhs()` (`pybamm.thermal.x_lumped.NoCurrentCollector method`), 63
`set_rhs()` (`pybamm.thermal.xyz_lumped.BaseModel method`), 65
`set_soc_variables()` (`pybamm.BaseBatteryModel method`), 26
`set_soc_variables()` (`pybamm.lead_acid.BaseModel method`), 27
`set_up()` (`pybamm.AlgebraicSolver method`), 84
`set_up()` (`pybamm.BaseSolver method`), 85
`set_up()` (`pybamm.DaeSolver method`), 87
`set_up()` (`pybamm.OdeSolver method`), 86
`set_variable_slices()` (`pybamm.Discretisation method`), 73
`shape` (`pybamm.Symbol attribute`), 6
`shape_for_testing` (`pybamm.Symbol attribute`), 6
`shift()` (`pybamm.FiniteVolume method`), 80
`Simplification` (class in `pybamm`), 20
`simplify()` (`pybamm.Simplification method`), 20
`simplify()` (`pybamm.Symbol method`), 6
`simplify_addition_subtraction()` (in module `pybamm`), 20
`simplify_if_constant()` (in module `pybamm`), 20
`simplify_multiplication_division()` (in module `pybamm`), 20
`Sin` (class in `pybamm`), 22
`Sinh` (class in `pybamm`), 22
`sinh()` (in module `pybamm`), 22
`size` (`pybamm.StateVector attribute`), 10
`size` (`pybamm.Symbol attribute`), 6
`size_for_testing` (`pybamm.Symbol attribute`), 6
`Solution` (class in `pybamm`), 89
`solve()` (`pybamm.AlgebraicSolver method`), 84
`solve()` (`pybamm.BaseSolver method`), 85
`source()` (in module `pybamm`), 12
`SparseStack` (class in `pybamm`), 19
`spatial_variable()` (`pybamm.FiniteVolume method`), 80
`spatial_variable()` (`pybamm.ScikitFiniteElement method`), 83
`spatial_variable()` (`pybamm.SpatialMethod method`), 77
`SpatialMethod` (class in `pybamm`), 74
`SpatialOperator` (class in `pybamm`), 13
`SpatialVariable` (class in `pybamm`), 8
`SpecificFunction` (class in `pybamm`), 21
`SPM` (class in `pybamm.lithium_ion`), 26

SPMe (class in *pybamm.lithium_ion*), 26
 StateVector (class in *pybamm*), 10
 step() (*pybamm.BaseSolver* method), 85
 stiffness_matrix() (*pybamm.ScikitFiniteElement* method), 83
 SubMesh0D (class in *pybamm*), 71
 SubMesh1D (class in *pybamm*), 71
 Subtraction (class in *pybamm*), 11
 surf() (in module *pybamm*), 17
 SurfaceForm (class in *pybamm.electrode.ohm*), 39
 Symbol (class in *pybamm*), 3

T

t (in module *pybamm*), 8
 t (*pybamm.Solution* attribute), 90
 t_event (*pybamm.Solution* attribute), 90
 termination (*pybamm.Solution* attribute), 90
 test_shape() (*pybamm.Symbol* method), 6
 Time (class in *pybamm*), 8
 time() (*pybamm.Timer* method), 92
 Timer (class in *pybamm*), 91

U

UnaryOperator (class in *pybamm*), 12
 Uniform (class in *pybamm.current_collector*), 31
 Uniform1DSubMesh (class in *pybamm*), 71
 unpack() (*pybamm.electrolyte.stefan_maxwell.conductivity.Composite* method), 41
 update() (*pybamm.BaseModel* method), 25
 update() (*pybamm.ParameterValues* method), 66
 update_model() (*pybamm.ParameterValues* method), 66
 update_scalars() (*pybamm.ParameterValues* method), 66
 use_jacobian (*pybamm.BaseModel* attribute), 24
 use_simplify (*pybamm.BaseModel* attribute), 24
 use_to_python (*pybamm.BaseModel* attribute), 24

V

value (*pybamm.Scalar* attribute), 9
 Variable (class in *pybamm*), 7
 variables (*pybamm.BaseModel* attribute), 23
 variables (*pybamm.BaseSubModel* attribute), 29
 Vector (class in *pybamm*), 9
 visualise() (*pybamm.Symbol* method), 6

X

x_average() (in module *pybamm*), 17

Y

y (*pybamm.Solution* attribute), 90
 y_event (*pybamm.Solution* attribute), 90

Z

ZeroDimensionalMethod (class in *pybamm*), 84