PyBaMM Documentation

Release 0.2.0

Valentin Sulzer

Mar 12, 2021

Contents

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

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

- genindex
- modindex
- search

CHAPTER 1

Contents

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)
- **auxiliary_domains** (*dict of str*) – dictionary of auxiliary domains over which the node is valid (empty dictionary indicates no auxiliary domains). Keys can be "secondary" or "tertiary". The symbol is broadcast over its auxiliary domains. For example, a symbol might have domain "negative particle", secondary domain "separator" and tertiary domain "current collector" (*domain="negative particle", auxiliary_domains={"secondary": "separator", "tertiary": "current collector"}*).

__abs__()

return an *AbsoluteValue* object

__add__(*other*)

return an *Addition* object

__ge__(*other*)

return a *Heaviside* object

__getitem__(*key*)

return a *Index* object

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

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

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

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

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

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

__neg__()
return a *Negate* object

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

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

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

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

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

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

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

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

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

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

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

children
returns the cached children of this node.

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

clear_domains()
Clear domains, bypassing checks

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

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

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

domain
list of applicable domains

Returns

Return type iterable of str

evaluate(*t=None*, *y=None*, *u=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)
- **u** (*dict, optional*) – dictionary of inputs to use when solving (default None)
- **known_evals** (*dict, optional*) – dictionary containing known values (default None)

Returns

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

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

evaluate_ignoring_errors()
Evaluates the expression. If a node exists in the tree that cannot be evaluated as a scalar or vector (e.g. Parameter, Variable, StateVector, InputParameter), 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 TyperError is raised. !Not to be confused with isinstance(self, pybamm.Scalar)!

See also:

evaluate() evaluate the expression

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

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

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

is_constant()
returns true if evaluating the expression is not dependent on *t* or *y* or *u*

See also:

evaluate() evaluate the expression

jac(*variable*, *known_jacs=None*)
Differentiate a symbol with respect to a (slice of) a State Vector. See *pybamm.Jacobian*.

name
name of the node

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

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

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

Examples

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

relabel_tree(*symbol*, *counter*)
Finds all children of a symbol and assigns them a new id so that they can be visualised properly using the graphviz output

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

secondary_domain
Helper function to get the secondary domain of a symbol

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

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

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

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

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

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

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

size_for_testing
Size of an object, based on shape for testing

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

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

to_casadi(*t=None*, *y=None*, *u=None*, *casadi_symbols=None*)
Convert the expression tree to a CasADi expression tree. See *pybamm.CasadiConverter*.

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

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

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

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()*.

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 represending a dependent variable

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

Parameters

- **name** (*str*) – name of the node
- **domain** (*iterable of str*) – list of domains that this variable is valid over
- **auxiliary_domains** (*dict*) – dictionary of auxiliary domains ({'secondary': ..., 'tertiary': ...}). For example, for the single particle model, the particle concentration would be a Variable with domain 'negative particle' and secondary auxiliary domain 'current collector'. For the DFN, the particle concentration would be a Variable with domain 'negative particle', secondary domain 'negative electrode' and tertiary domain 'current collector'
- ***Extends** –

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

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

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

Parameters

- **name** (*str*) – name of the node
- **domain** (*iterable of str*) – list of domains that this variable is valid over
- **auxiliary_domains** (*dict*) – dictionary of auxiliary domains ({'secondary': ..., 'tertiary': ...}). For example, for the single particle model, the particle concentration would be a Variable with domain 'negative particle' and secondary auxiliary domain 'current collector'. For the DFN, the particle concentration would be a Variable with domain 'negative particle', secondary domain 'negative electrode' and tertiary domain 'current collector'
- ***Extends** –

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

1.1.4 Independent Variable

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

Used for expressing functions depending on a spatial variable or time

Parameters

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

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

Extends: *Symbol*

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

class pybamm.**SpatialVariable**(*name*, *domain=None*, *auxiliary_domains=None*, *coord_sys=None*)
A node in the expression tree representing a spatial variable

Parameters

- **name** (*str*) – name of the node (e.g. "x", "y", "z", "r", "x_n", "x_s", "x_p", "r_n", "r_p")
- **domain** (*iterable of str*) – list of domains that this variable is valid over (e.g. "cartesian", "spherical polar")
- ***Extends** –

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

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

Extends: *Symbol*

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

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

Extends: `Array`

1.1.7 Vector

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

Extends: `Array`

1.1.8 State Vector

class pybamm.**StateVector**(**y_slices*, *name=None*, *domain=None*, *auxiliary_domains=None*, *evaluation_array=None*)
node in the expression tree that holds a slice to read from an external vector type

Parameters

- **y_slice** (*slice*) – the slice of an external y to read
- **name** (*str, optional*) – the name of the node
- **domain** (*iterable of str, optional*) – list of domains the parameter is valid over, defaults to empty list
- **auxiliary_domains** (*dict of str, optional*) – dictionary of auxiliary domains
- **evaluation_array** (*list, optional*) – List of boolean arrays representing slices. Default is None, in which case the evaluation_array is computed from y_slices.
- ***Extends** –

evaluation_array
Array to use for evaluating

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

set_evaluation_array(*y_slices*, *evaluation_array*)
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*, *u=None*, *known_evals=None*)
See *pybamm.Symbol.evaluate()*.

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

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

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

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

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

Extends: *BinaryOperator*

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

Extends: *BinaryOperator*

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

Extends: *BinaryOperator*

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

Extends: *BinaryOperator*

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

Extends: *BinaryOperator*

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

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

Extends: *BinaryOperator*

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

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

Extends: *BinaryOperator*

evaluates_on_edges()

See *pybamm.Symbol.evaluates_on_edges()*.

class pybamm.**Heaviside**(*left*, *right*, *equal*)

A node in the expression tree representing a heaviside step function.

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

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

Extends: *BinaryOperator*

diff(*variable*)

See *pybamm.Symbol.diff()*.

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

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

Parameters

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

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

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

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

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

Extends: *UnaryOperator*

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

Extends: *UnaryOperator*

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

class pybamm.**Index**(*child*, *index*, *name=None*, *check_size=True*)
A node in the expression tree, which stores the index that should be extracted from its child after the child has been evaluated.

Parameters

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

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

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

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

Derived classes will specify the particular operator

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

Extends: *UnaryOperator*

Parameters

- **name** (*str*) – name of the node

- **child** (*Symbol*) – child node

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

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

Extends: *SpatialOperator*

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

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

Extends: *SpatialOperator*

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

class pybamm.**Laplacian**(*child*)
A node in the expression tree representing a laplacian operator. This is currently only implemeted in the weak form for finite element formulations.

Extends: *SpatialOperator*

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

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

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

class pybamm.**Mass**(*child*)
Returns the mass matrix for a given symbol, accounting for Dirchlet boundary conditions where necessary (e.g. in the finite element formualtion) **Extends:** *SpatialOperator*

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

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

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

Parameters

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

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

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

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

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

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

Parameters

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

class pybamm.**DefiniteIntegralVector**(*child*, *vector_type='row'*)
A node in the expression tree representing an integral of the basis used for discretisation

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

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

Parameters

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

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

class pybamm.**BoundaryIntegral**(*child*, *region='entire'*)
A node in the expression tree representing an integral operator over the boundary of a domain

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

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

Parameters

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

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

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

class pybamm.**DeltaFunction**(*child*, *side*, *domain*)
Delta function. Currently can only be implemented at the edge of a domain

Parameters

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

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

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

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

Parameters

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

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

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

Parameters

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

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

Parameters

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

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

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

Returns the gradient of `expression`

Return type *Gradient*

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

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

Returns the divergence of `expression`

Return type *Divergence*

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

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

Returns the laplacian of `expression`

Return type *Laplacian*

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

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

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

Return type *Gradient_Squared*

pybamm.**surf**(*symbol*)
convenience function for creating a right *BoundaryValue*, usually in the spherical geometry

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

Returns the surface value of `symbol`

Return type *pybamm.BoundaryValue*

pybamm.**x_average**(*symbol*)
convenience function for creating an average in the x-direction

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

Returns the new averaged symbol

Return type *Symbol*

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

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

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

Upon evaluation, equations are concatenated using numpy concatenation.

Extends: *Concatenation*

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

class pybamm.**DomainConcatenation**(*children*, *full_mesh*, *copy_this=None*)
A node in the expression tree representing a concatenation of symbols, being careful about domains.

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

Extends: *pybamm.Concatenation*

Parameters

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

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

Extends: *Concatenation*

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

1.1.12 Broadcasting Operators

class pybamm.**Broadcast**(*child*, *broadcast_domain*, *broadcast_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.

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

Parameters

- **child** (*Symbol*) – child node
- **broadcast_domain** (*iterable of str*) – Primary domain for broadcast. This will become the domain of the symbol
- **broadcast_auxiliary_domains** (*dict of str*) – Auxiliary domains for broadcast.

- **broadcast_type** (*str, optional*) – Whether to broadcast to the full domain (primary and secondary) or only in the primary direction. Default is "full".
- **name** (*str*) – name of the node
- ****Extends** (** *SpatialOperator*) –

class pybamm.**FullBroadcast**(*child*, *broadcast_domain*, *auxiliary_domains*, *name=None*)
A class for full broadcasts

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

class pybamm.**PrimaryBroadcast**(*child*, *broadcast_domain*, *name=None*)
A node in the expression tree representing a primary broadcasting operator. Broadcasts in a *primary* dimension only. That is, makes explicit copies of the symbol in the domain specified by *broadcast_domain*. This should be used for broadcasting from a "larger" scale to a "smaller" scale, for example broadcasting temperature T(x) from the electrode to the particles, or broadcasting current collector current i(y, z) from the current collector to the electrodes.

Parameters

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

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

class pybamm.**SecondaryBroadcast**(*child*, *broadcast_domain*, *name=None*)
A node in the expression tree representing a primary broadcasting operator. Broadcasts in a *secondary* dimension only. That is, makes explicit copies of the symbol in the domain specified by *broadcast_domain*. This should be used for broadcasting from a "smaller" scale to a "larger" scale, for example broadcasting SPM particle concentrations c_s(r) from the particles to the electrodes. Note that this wouldn't be used to broadcast particle concentrations in the DFN, since these already depend on both x and r.

Parameters

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

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

1.1.13 Functions

class pybamm.**Function**(*function*, **children*, *name=None*, *derivative='autograd'*, *differentiated_function=None*)
A node in the expression tree representing an arbitrary function

Parameters

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

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

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

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

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

class pybamm.**SpecificFunction**(*function*, *child*)
Parent class for the specific functions, which implement their own *diff* operators directly.

Parameters

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

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

pybamm.**cos**(*child*)
Returns cosine function of child.

class pybamm.**Cosh**(*child*)
Hyberbolic cosine function

pybamm.**cosh**(*child*)
Returns hyperbolic cosine function of child.

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

pybamm.**exp**(*child*)
Returns exponential function of child.

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

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

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

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.

1.1.14 Input Parameter

class pybamm.**InputParameter**(*name*)
A node in the expression tree representing an input parameter

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

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

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

1.1.15 Interpolant

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

Parameters

- **data** (numpy.ndarray) – Numpy array of data to use for interpolation. Must have exactly two columns (x and y data)
- **child** (*pybamm.Symbol*) – Node to use when evaluating the interpolant
- **name** (*str, optional*) – Name of the interpolant. Default is None, in which case the name "interpolating function" is given.
- **interpolator** (*str, optional*) – Which interpolator to use ("pchip" or "cubic spline"). Note that whichever interpolator is used must be differentiable (for Interpolator._diff). Default is "cubic spline". Note that "pchip" may give slow results.
- **extrapolate** (*bool, optional*) – Whether to extrapolate for points that are outside of the parametrisation range, or return NaN (following default behaviour from scipy). Default is True.
- ****Extends**** (*pybamm.Function*) –

1.1.16 Operations on expression trees

Classes and functions that operate on the expression tree

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

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

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

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

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

Parameters

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

pybamm.**simplify_multiplication_division**(*myclass*, *left*, *right*)
if children are associative (multiply, division, etc) then try to find groups of constant children (that produce a value) and simplify them

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

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

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

Parameters

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

EvaluatorPython

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

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

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

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

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

Jacobian

class pybamm.**Jacobian**(*known_jacs=None*)

jac(*symbol*, *variable*)

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

Parameters

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

Returns Symbol representing the Jacobian

Return type *pybamm.Symbol*

Convert to CasADi

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

convert(*symbol*, *t=None*, *y=None*, *u=None*)

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

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to convert

- **t** (casadi.MX) – A casadi symbol representing time
- **y** (casadi.MX) – A casadi symbol representing state vectors
- **u** (*dict*) – A dictionary of casadi symbols representing inputs

Returns The converted symbol

Return type casadi.MX

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

Type list of *pybamm.Event*

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

Type *pybamm.Concatenation*

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

Type *pybamm.Concatenation*

concatenated_initial_conditions
After discretisation, contains the vector of initial conditions

Type `numpy.array`

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

Type *pybamm.Matrix*

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

Type *pybamm.Matrix*

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

Type *pybamm.Concatenation*

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

Type *pybamm.Concatenation*

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

Type *pybamm.Concatenation*

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

Type bool

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

Type bool

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

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

Default is "casadi".

Type str

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

check_default_variables_dictionaries()
Chec that the right variables are provided.

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

check_well_determined(*post_discretisation*)
Check that the model is not under- or over-determined.

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

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

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

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

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

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

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

Base Battery Model

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

Base model class with some default settings and required variables

options

A dictionary of options to be passed to the model. The options that can be set are listed below. Note that not all of the options are compatible with each other and with all of the models implemented in PyBaMM.

- **"dimensionality"** [int, optional] Sets the dimension of the current collector problem. Can be 0 (default), 1 or 2.
- **"surface form"** [bool or str, optional] Whether to use the surface formulation of the problem. Can be False (default), "differential" or "algebraic". 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 "set external potential". The submodel "set external potential" can only be used with the SPM.
- **"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", "lumped" or "set external temperature". Must be "isothermal" for lead-acid models. If the option "set external temperature" is selected then "dimensionality" must be 1.
- **"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.
- **"external submodels"** [list] A list of the submodels that you would like to supply an external variable for instead of solving in PyBaMM. The entries of the lists are strings that correspond to the submodel names in the keys of *self.submodels*.

Type dict

Extends: *pybamm.BaseModel*

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

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

Parameters

- **symbol** (*pybamm.Symbol*) – Symbol to be processed

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

Returns Processed symbol

Return type *pybamm.Symbol*

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

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

Event

class pybamm.**Event**(*name*, *expression*, *event_type=<EventType.TERMINATION: 0>*)
Defines an event for use within a pybamm model

name
A string giving the name of the event

Type str

event_type
An enum defining the type of event

Type *pybamm.EventType*

expression
An expression that defines when the event occurs

Type *pybamm.Symbol*

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

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

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

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

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'*, *build=True*)
Single Particle Model (SPM) of a lithium-ion battery, from[1].

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the model.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

References

Extends: *pybamm.lithium_ion.BaseModel*

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

This class differs from the *pybamm.lithium_ion.SPM* model class in that it shows the whole model in a single class. This comes at the cost of flexibility in combining different physical effects, and in general the main SPM class should be used instead.

Parameters **name** (*str, optional*) – The name of the model.

References

Extends: *pybamm.lithium_ion.BaseModel*

Single Particle Model with Electrolyte (SPMe)

class pybamm.lithium_ion.**SPMe**(*options=None*, *name='Single Particle Model with electrolyte'*, *build=True*)
Single Particle Model with Electrolyte (SPMe) of a lithium-ion battery, from[1].

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the model.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

References

Extends: *pybamm.lithium_ion.BaseModel*

[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).
[2] SG Marquis, V Sulzer, R Timms, CP Please and SJ Chapman. "An asymptotic derivation of a single particle model with electrolyte". In: arXiv preprint arXiv:1905.12553 (2019).
[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'*, *build=True*)
Doyle-Fuller-Newman (DFN) model of a lithium-ion battery, from[1].

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the model.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

References

Extends: *pybamm.lithium_ion.BaseModel*

class pybamm.lithium_ion.**BasicDFN**(*name='Doyle-Fuller-Newman model'*)
Doyle-Fuller-Newman (DFN) model of a lithium-ion battery, from[2].

This class differs from the *pybamm.lithium_ion.DFN* model class in that it shows the whole model in a single class. This comes at the cost of flexibility in comparing different physical effects, and in general the main DFN class should be used instead.

Parameters **name** (*str, optional*) – The name of the model.

References

Extends: *pybamm.lithium_ion.BaseModel*

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*

default_solver
Return default solver based on whether model is ODE model or DAE model. There are bugs with KLU on the lead-acid models.

set_soc_variables()
Set variables relating to the state of charge.

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

Leading-Order Quasi-Static Model

class pybamm.lead_acid.**LOQS**(*options=None*, *name='LOQS model'*, *build=True*)

Leading-Order Quasi-Static model for lead-acid, from[1].

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the model.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

References

Extends: *pybamm.lead_acid.BaseModel*

set_external_circuit_submodel()

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

Higher-Order Models

class pybamm.lead_acid.**BaseHigherOrderModel**(*options=None*, *name='Composite model'*, *build=True*)

Base model for higher-order models for lead-acid, from[1]. Uses leading-order model from *pybamm.lead_acid.LOQS*

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the model.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

References

Extends: *pybamm.lead_acid.BaseModel*

set_full_convection_submodel()

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

set_full_interface_submodel()

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

set_full_porosity_submodel()

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

[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.

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

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the model.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).
- ****Extends** (** *pybamm.lead_acid.BaseHigherOrderModel*) –

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

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

Extends: *pybamm.lead_acid.BaseHigherOrderModel*

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

class pybamm.lead_acid.**CompositeExtended**(*options=None*, *name='Extended composite model'*, *build=True*)
Extended composite model for lead-acid, from[2]. Uses leading-order model from *pybamm.lead_acid.LOQS*

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the model.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

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'*, *build=True*)
Porous electrode model for lead-acid, from[1], based on the Full model.

Parameters

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

- **name** (*str, optional*) – The name of the model.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

References

Extends: *pybamm.lead_acid.BaseModel*

1.2.4 Submodels

Base Submodel

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

The base class for all submodels. All submodels inherit from this class and must only provide public methods which overwrite those in this base class. Any methods added to a submodel that do not overwrite those in this bass class are made private with the prefix '_', providing a consistent public interface for all submodels.

Parameters **param** (*parameter class*) – The model parameter symbols

param
The model parameter symbols

Type parameter class

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

Type dict

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

Type dict

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

Type dict

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

Type dict

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

Type dict

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

Type list

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

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

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

Return type dict

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

Returns A list of the external variables in the model.

Return type list

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

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

Return type dict

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

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

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

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

set_events(*variables*)
A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used a implemented in *pybamm.BaseSubModel*.

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

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

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

set_rhs(*variables*)
A method to set the right hand side of the differential equations which contain a time derivative. Note: this

method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

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

Current Collector

Base Model

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

Base class for current collector submodels

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

Extends: *pybamm.BaseSubModel*

Composite Potential Pair models

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

Composite potential pair model for the current collectors. This is identical to the BasePotentialPair model, except the name of the fundamental variables are changed to avoid clashes with leading order.

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

Extends: *pybamm.current_collector.BasePotentialPair*

get_fundamental_variables()

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

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

Return type dict

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

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

Effective Current collector Resistance models

class pybamm.current_collector.**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*

default_solver

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

get_processed_potentials(*solution*, *param_values*, *V_av*, *I_av*)

Calculates the potentials in the current collector given the average voltage and current. Note: This takes in the *processed* V_av and I_av from a 1D simulation representing the average cell behaviour and returns a dictionary of processed potentials.

Uniform

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

A submodel for uniform potential in the current collectors which is valid in the limit of fast conductivity in the current collectors.

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

Extends: *pybamm.current_collector.BaseModel*

get_coupled_variables(*variables*)

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

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

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

Return type dict

Potential Pair models

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

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

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

Extends: *pybamm.current_collector.BaseModel*

get_fundamental_variables()

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

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

Return type dict

set_algebraic(*variables*)

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

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

set_initial_conditions(*variables*)

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

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

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

Base class for a 2+1D potential pair model

set_boundary_conditions(*variables*)

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

self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used a implemented in *pybamm.BaseSubModel*.

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

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

Base class for a 1+1D potential pair model.

set_boundary_conditions(*variables*)

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

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

Quite Conductive Potential Pair models

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

A submodel for Ohm's law plus conservation of current in the current collectors, in the limit of quite conductive electrodes.

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

Extends: *pybamm.current_collector.BaseModel*

get_fundamental_variables()

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

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

Return type dict

set_algebraic(*variables*)

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

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

set_initial_conditions(*variables*)

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

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

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

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

Set Potential Single Particle Models

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

A submodel for current collectors which *doesn't* update the potentials during solve. This class uses the current-

voltage relationship from the SPM(e) (see[1]) to calculate the current.

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

References

Extends: *pybamm.current_collector.BaseModel*

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

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

Return type dict

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

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

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

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

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

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

class pybamm.current_collector.**SetPotentialSingleParticle1plus1D**(*param*)
Class for 1+1D set potential model

class pybamm.current_collector.**SetPotentialSingleParticle2plus1D**(*param*)
Class for 1+1D set potential model

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*

[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).

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 a implemented in *pybamm.BaseSubModel*.

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

set_initial_conditions(*variables*)

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

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

Electrode

Electrode Base Model

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

Base class for electrode submodels.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'Negative' or 'Positive'
- **set_positive_potential** (*bool, optional*) – If True the battery model sets the positve potential based on the current. If False, the potential is specified by the user. Default is True.
- ****Extends** (** *pybamm.BaseSubModel*) –

Ohmic

Base Model

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

A base class for electrode submodels that employ Ohm's law.

Parameters

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

Extends: *pybamm.electrode.BaseElectrode*

set_boundary_conditions(*variables*)

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

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

Leading Order Model

class pybamm.electrode.ohm.**LeadingOrder**(*param*, *domain*, *set_positive_potential=True*)

An electrode submodel that employs Ohm's law the leading-order approximation to governing equations.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'Negative' or 'Positive'
- **set_positive_potential** (*bool, optional*) – If True the battery model sets the positve potential based on the current. If False, the potential is specified by the user. Default is True.
- ****Extends** (** *pybamm.electrode.ohm.BaseModel*) –

get_coupled_variables(*variables*)

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

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

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

Composite Model

class pybamm.electrode.ohm.**Composite**(*param*, *domain*)
An explicit composite leading and first order solution to solid phase current conservation with ohm's law. Note that the returned current density is only the leading order approximation.

Parameters

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

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

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

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

Return type dict

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

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

Full Model

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

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

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

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

Return type dict

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

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

Return type dict

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

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

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

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

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

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

Surface Form

class pybamm.electrode.ohm.**SurfaceForm**(*param*, *domain*)
A submodel for the electrode with Ohm's law in the surface potential formulation.

Parameters

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

Extends: *pybamm.electrode.ohm.BaseModel*

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

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

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

Return type dict

Electrolyte

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*, *domain=None*, *reactions=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*, *domain=None*, *reactions=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=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.BaseHigerOrder) –

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) –

get_coupled_variables(*variables*)

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

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

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

Return type dict

get_external_variables()

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

Returns A list of the external variables in the model.

Return type list

get_fundamental_variables()

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

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

Return type dict

set_algebraic(*variables*)

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

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

set_boundary_conditions(*variables*)

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

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

set_events(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used a implemented in *pybamm.BaseSubModel*.

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

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

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

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

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

Surface Form

Full Model

class pybamm.electrolyte.stefan_maxwell.conductivity.surface_potential_form.**FullDifferentia**

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**(

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.**LeadingOrderDi**

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.**LeadingOrderAlg**

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 a 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 a 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 a implemented in *pybamm.BaseSubModel*.

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

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

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

Leading Order Model

class pybamm.electrolyte.stefan_maxwell.diffusion.**LeadingOrder** (*param*, *reactions*)
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 a implemented in *pybamm.BaseSubModel*.

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

set_rhs(*variables*)

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

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

External circuit

Models to enforce different boundary conditions (as imposed by an imaginary external circuit) such as constant current, constant voltage, constant power, or any other relationship between the current and voltage. "Current control" enforces these directly through boundary conditions, while "Function control" submodels add an algebraic equation (for the current) and hence can be used to set any variable to be constant.

Current control external circuit

class pybamm.external_circuit.**CurrentControl**(*param*)

External circuit with current control.

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

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

Return type dict

Function control external circuit

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

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

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

Return type dict

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

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

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

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

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

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

Interface

Interface Base Model

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

Inverse Interface Kinetics

Base Inverse First-order Kinetics

class pybamm.interface.inverse_kinetics.**BaseInverseFirstOrderKinetics**(*param*, *domain*)

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:

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

Parameters

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

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:

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

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 a implemented in *pybamm.BaseSubModel*.

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

set_initial_conditions(*variables*)

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

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

set_rhs(*variables*)

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

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

Leading Order Model

class pybamm.oxygen_diffusion.**LeadingOrder**(*param*, *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.oxgen_diffusion.BaseModel) –

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

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

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

Return type dict

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

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

Return type dict

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

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

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

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

No Oxygen

class pybamm.oxygen_diffusion.**NoOxygen**(*param*)
Class for when there is no oxygen

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

Extends: *pybamm.oxygen_diffusion.BaseModel*

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

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

Return type dict

Particle

Particle Base Model

class pybamm.particle.**BaseParticle**(*param*, *domain*)
Base class for molar conservation in particles.

Parameters

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

Extends: *pybamm.BaseSubModel*

set_events(*variables*)
A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used a implemented in *pybamm.BaseSubModel*.

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

Fickian

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.BaseParticle*

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

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

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

Return type dict

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

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

Return type dict

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

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

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

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

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

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

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.BaseParticle*

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

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

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

Return type dict

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

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

Return type dict

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

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

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

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

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

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

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.

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

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

Porosity

Base Model

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

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

Extends: *pybamm.BaseSubModel*

set_events(*variables*)
A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used a implemented in *pybamm.BaseSubModel*.

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

Constant Porosity

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

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

Extends: *pybamm.porosity.BaseModel*

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

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

Return type dict

Leading-Order Model

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

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

Extends: *pybamm.porosity.BaseModel*

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

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

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

Return type dict

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

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

Return type dict

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

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

set_rhs(*variables*)
A method to set the right hand side of the differential equations which contain a time derivative. Note: this

method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

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

Full Model

class pybamm.porosity.**Full**(*param*)

Full model for reaction-driven porosity changes

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

Extends: *pybamm.porosity.BaseModel*

get_coupled_variables(*variables*)

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

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

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

Return type dict

get_fundamental_variables()

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

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

Return type dict

set_initial_conditions(*variables*)

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

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

set_rhs(*variables*)

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

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

Thermal

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 a 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 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.

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 a 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 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.

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 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.

Set Temperature 1D current collector

class pybamm.thermal.x_lumped.**SetTemperature1D**(*param*)

Class for x-lumped thermal submodel which *doesn't* update the temperature. Instead, the temperature can be set (as a function of space) externally. Note, this model computes the heat generation terms for inspection after solve.

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.

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 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.

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*

Tortuosity

Base Model

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

Parameters

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

Bruggeman Model

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

Extends: *pybamm.tortuosity.BaseModel*

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

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

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

Return type dict

1.3 Parameters

1.3.1 Base Parameter Values

class pybamm.**ParameterValues**(*values=None*, *chemistry=None*)
The parameter values for a simulation.

Note that this class does not inherit directly from the python dictionary class as this causes issues with saving and loading simulations.

Parameters

- **values** (*dict or string*) – Explicit set of parameters, or reference to a file of parameters If string, gets passed to read_parameters_csv to read a file.
- **chemistry** (*dict*) – Dict of strings for default chemistries. Must be of the form: {"base chemistry": base_chemistry, "cell": cell_properties_authorYear, "anode": anode_chemistry_authorYear, "separator": separator_chemistry_authorYear, "cathode": cathode_chemistry_authorYear, "electrolyte": electrolyte_chemistry_authorYear, "experiment": experimental_conditions_authorYear}. Then the anode chemistry is loaded from the file inputs/parameters/base_chemistry/anodes/anode_chemistry_authorYear, etc. Parameters in "cell" should include geometry and current collector properties. Parameters in "experiment"

should include parameters relating to experimental conditions, such as initial conditions and currents.

Examples

```
>>> import pybamm
>>> values = {"some parameter": 1, "another parameter": 2}
>>> param = pybamm.ParameterValues(values)
>>> param["some parameter"]
1
>>> file = "input/parameters/lithium-ion/cells/kokam_Marquis2019/parameters.csv"
>>> values_path = pybamm.get_parameters_filepath(file)
>>> param = pybamm.ParameterValues(values=values_path)
>>> param["Negative current collector thickness [m]"]
2.5e-05
>>> param = pybamm.ParameterValues(chemistry=pybamm.parameter_sets.Marquis2019)
>>> param["Reference temperature [K]"]
298.15
```

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

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

Returns The evaluated symbol

Return type number of array

items ()
Get the items of the dictionary

keys ()
Get the keys of the dictionary

process_boundary_conditions (*model*)
Process boundary conditions for a model Boundary conditions are dictionaries {"left": left bc, "right": right bc} in general, but may be imposed on the tabs (or *not* on the tab) for a small number of variables, e.g. {"negative tab": neg. tab bc, "positive tab": pos. tab bc "no tab": no tab bc}.

process_geometry (*geometry*)
Assign parameter values to a geometry (inplace).

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

process_model (*unprocessed_model*, *inplace=True*)
Assign parameter values to a model. Currently inplace, could be changed to return a new model.

Parameters

- **unprocessed_model** (*pybamm.BaseModel*) – Model to assign parameter values for
- **inplace** (*bool, optional*) – If True, replace the parameters in the model in place. Otherwise, return a new model with parameter values set. Default is True.

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

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 (*values*, *check_conflict=False*, *check_already_exists=True*, *path=''*)
Update parameter dictionary, while also performing some basic checks.

Parameters

- **values** (*dict*) – Dictionary of parameter values to update parameter dictionary with
- **check_conflict** (*bool, optional*) – Whether to check that a parameter in *values* has not already been defined in the parameter class when updating it, and if so that its value does not change. This is set to True during initialisation, when parameters are combined from different sources, and is False by default otherwise
- **check_already_exists** (*bool, optional*) – Whether to check that a parameter in *values* already exists when trying to update it. This is to avoid cases where an intended change in the parameters is ignored due a typo in the parameter name, and is True by default but can be manually overridden.
- **path** (*string, optional*) – Path from which to load functions

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

values ()
Get the values of the dictionary

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 * 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

1.3.8 Parameters Sets

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

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": l_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.**Geometryxp1p1DMicro**(*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

class pybamm.**SubMesh**

Base submesh class. Contains the position of the nodes, the number of mesh points, and (optionally) information about the tab locations.

class pybamm.**MeshGenerator**(*submesh_type*, *submesh_params=None*)

Base class for mesh generator objects that are used to generate submeshes.

Parameters

- **submesh_type** (*pybamm.SubMesh*) – The type of submesh to use (e.g. Uniform1DSubMesh).
- **submesh_params** (*dict, optional*) – Contains any parameters required by the submesh.

1.5.2 0D Sub Mesh

class pybamm.**SubMesh0D**(*position*, *npts=None*, *tabs=None*)
0D submesh class. Contains the position of the node.

Parameters

- **position** (*dict*) – A dictionary that contains the position of the 0D submesh (a signle point) in space
- **npts** (*dict, optional*) – Number of points to be used. Included for compatibility with other meshes, but ignored by this mesh class
- **tabs** (*dict*) – A dictionary that contains information about the size and location of the tabs. Included for compatibility with other meshes, but ignored by this mesh class
- ****Extends** (“”: *pybamm.SubMesh*) –

1.5.3 1D Sub Meshes

class pybamm.**SubMesh1D**(*edges*, *coord_sys*, *tabs=None*)
1D submesh class. Contains the position of the nodes, the number of mesh points, and (optionally) information about the tab locations.

Parameters

- **edges** (*array_like*) – An array containing the points corresponding to the edges of the submesh
- **coord_sys** (*string*) – The coordinate system of the submesh
- **tabs** (*dict, optional*) – A dictionary that contains information about the size and location of the tabs
- ****Extends** (“”: *pybamm.SubMesh*) –

class pybamm.**Uniform1DSubMesh**(*lims*, *npts*, *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. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.
- **tabs** (*dict, optional*) – A dictionary that contains information about the size and location of the tabs
- ****Extends** (“”: *pybamm.SubMesh1D*) –

class pybamm.**Exponential1DSubMesh**(*lims*, *npts*, *tabs*, *side='symmetric'*, *stretch=None*)
A class to generate a submesh on a 1D domain in which the points are clustered close to one or both of boundaries using an exponential formula on the interval [a,b].

If side is “left”, the gridpoints are given by

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

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

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

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

+ a,for k = 1, . . . , N. The grid spacing is then reflected to contruct the grid on the full interval [a,b].

In the above, alpha is a stretching factor. As the number of gridpoints tends to infinity, the ratio of the largest and smallest grid cells tends to exp(alpha).

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.
- **tabs** (*dict*) – A dictionary that contains information about the size and location of the tabs
- **side** (*str, optional*) – Whether the points are clustered near to the left or right boundary, or both boundaries. Can be “left”, “right” or “symmetric”. Default is “symmetric”
- **stretch** (*float, optional*) – The factor (alpha) which appears in the exponential. If side is “symmetric” then the default stretch is 1.15. If side is “left” or “right” then the default stretch is 2.3.
- ****Extends** (“”: *pybamm.SubMesh1D*) –

class pybamm.**Chebyshev1DSubMesh**(*lims*, *npts*, *tabs=None*)

A class to generate a submesh on a 1D domain using Chebyshev nodes on the interval (a, b), given by

$$x_k = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos(\frac{2k-1}{2N}\pi),$$

for k = 1, . . . , N, where N is the number of nodes. Note: this mesh then appends the boundary edges, so that the mesh edges are given by

$$a < x_1 < ... < x_N < b.$$

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.
- **tabs** (*dict, optional*) – A dictionary that contains information about the size and location of the tabs
- ****Extends** (“”: *pybamm.SubMesh1D*) –

class pybamm.**UserSupplied1DSubMesh**(*lims*, *npts*, *tabs*, *edges=None*)

A class to generate a submesh on a 1D domain from a user supplied array of edges.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.

- **tabs** (*dict*) – A dictionary that contains information about the size and location of the tabs
- **edges** (*array_like*) – The array of points which correspond to the edges of the mesh.
- ****Extends** ("": *pybamm.SubMesh1D*) –

1.5.4 2D Sub Meshes

class pybamm.**ScikitSubMesh2D**(*edges*, *coord_sys*, *tabs*)

2D submesh class. Contains information about the 2D finite element mesh. Note: This class only allows for the use of piecewise-linear triangular finite elements.

Parameters

- **edges** (*array_like*) – An array containing the points corresponding to the edges of the submesh
- **coord_sys** (*string*) – The coordinate system of the submesh
- **tabs** (*dict, optional*) – A dictionary that contains information about the size and location of the tabs
- ****Extends** ("": *pybamm.SubMesh*) –

on_boundary(*y*, *z*, *tab*)

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

class pybamm.**ScikitUniform2DSubMesh**(*lims*, *npts*, *tabs*)

Contains information about the 2D finite element mesh with uniform grid spacing (can be different spacing in y and z). Note: This class only allows for the use of piecewise-linear triangular finite elements.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of each spatial variable
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable
- **tabs** (*dict*) – A dictionary that contains information about the size and location of the tabs
- ****Extends** ("": *pybamm.ScikitSubMesh2D*) –

class pybamm.**ScikitExponential2DSubMesh**(*lims*, *npts*, *tabs*, *side='top'*, *stretch=2.3*)

Contains information about the 2D finite element mesh generated by taking the tensor product of a uniformly spaced grid in the y direction, and a unequally spaced grid in the z direction in which the points are clustered close to the top boundary using an exponential formula on the interval [a,b]. The gridpoints in the z direction are given by

+ a,for k = 1, . . . , N, where N is the number of nodes. Here alpha is a stretching factor. As the number of gridpoints tends to infinity, the ratio of the largest and smallest grid cells tends to exp(alpha).

Note: in the future this will be extended to allow points to be clustered near any of the boundaries.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of each spatial variable
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable
- **tabs** (*dict*) – A dictionary that contains information about the size and location of the tabs

- **side** (*str, optional*) – Whether the points are clustered near to a particular boundary. At present, can only be "top". Default is "top".
- **stretch** (*float, optional*) – The factor (alpha) which appears in the exponential. Default is 2.3.
- ****Extends** ("": *pybamm.ScikitSubMesh2D*) –

class pybamm.**ScikitChebyshev2DSubMesh**(*lims*, *npts*, *tabs*)

Contains information about the 2D finite element mesh generated by taking the tensor product of two 1D meshes which use Chebyshev nodes on the interval (a, b), given by

$$x_k = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos(\frac{2k-1}{2N}\pi),$$

for k = 1, ..., N, where N is the number of nodes. Note: this mesh then appends the boundary edgess, so that the 1D mesh edges are given by

$$a < x_1 < ... < x_N < b.$$

Note: This class only allows for the use of piecewise-linear triangular finite elements.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of each spatial variable
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable
- **tabs** (*dict*) – A dictionary that contains information about the size and location of the tabs
- ****Extends** ("": *pybamm.ScikitSubMesh2D*) –

class pybamm.**UserSupplied2DSubMesh**(*lims*, *npts*, *tabs*, *y_edges=None*, *z_edges=None*)

A class to generate a tensor product submesh on a 2D domain by using two user supplied vectors of edges: one for the y-direction and one for the z-direction. Note: this mesh should be created using `UserSupplied2DSubMeshGenerator`.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.
- **tabs** (*dict*) – A dictionary that contains information about the size and location of the tabs
- **y_edges** (*array_like*) – The array of points which correspond to the edges in the y direction of the mesh.
- **z_edges** (*array_like*) – The array of points which correspond to the edges in the z direction of the mesh.
- ****Extends** ("": *pybamm.ScikitSubMesh2D*) –

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 methods to be used on each domain. The keys correspond to the model domains and the values to the spatial method.

check_initial_conditions(*model*)

Check initial conditions are a numpy array

check_initial_conditions_rhs(*model*)

Check initial conditions and rhs have the same shape

check_model(*model*)

Perform some basic checks to make sure the discretised model makes sense.

check_tab_conditions(*symbol*, *bcs*)

Check any boundary conditions applied on "negative tab", "positive tab" and "no tab". For 1D current collector meshes, these conditions are converted into boundary conditions on "left" (tab at z=0) or "right" (tab at z=l_z) depending on the tab location stored in the mesh. For 2D current collector meshes, the boundary conditions can be applied on the tabs directly.

Parameters

- **symbol** (pybamm.expression_tree.symbol.Symbol) – The symbol on which the boundary conditions are applied.
- **bcs** (*dict*) – The dictionary of boundary conditions (a dict of {side: equation}).

Returns The dictionary of boundary conditions, with the keys changed to "left" and "right" where necessary.

Return type dict

check_variables(*model*)

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

create_jacobian(*model*)

Creates Jacobian of the discretised model. Note that the model is assumed to be of the form M*y_dot = f(t,y), where M is the (possibly singular) mass matrix. The Jacobian is df/dy.

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

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

Returns The expression trees corresponding to the Jacobian of the model

Return type *pybamm.Concatenation*

create_mass_matrix(*model*)
Creates mass matrix of the discretised model. Note that the model is assumed to be of the form M*y_dot = f(t,y), where M is the (possibly singular) mass matrix.

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

Returns

- *pybamm.Matrix* – The mass matrix
- *pybamm.Matrix* – The inverse of the ode part of the mass matrix (required by solvers which only accept the ODEs in explicit form)

process_boundary_conditions(*model*)
Discretise model boundary_conditions, also converting keys to ids

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

Returns Dictionary of processed boundary conditions

Return type dict

process_dict(*var_eqn_dict*)
Discretise a dictionary of {variable: equation}, broadcasting if necessary (can be model.rhs, model.algebraic, model.initial_conditions or model.variables).

Parameters **var_eqn_dict** (*dict*) – Equations ({variable: equation} dict) to dicretise (can be model.rhs, model.algebraic, model.initial_conditions or model.variables)

Returns **new_var_eqn_dict** – Discretised equations

Return type dict

process_initial_conditions(*model*)
Discretise model initial_conditions.

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

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

Return type tuple

process_model(*model*, *inplace=True*, *check_model=True*)
Discretise a model. Currently inplace, could be changed to return a new model.

Parameters

- **model** (*pybamm.BaseModel*) – Model to dicretise. Must have attributes rhs, initial_conditions and boundary_conditions (all dicts of {variable: equation})
- **inplace** (*bool, optional*) – If True, discretise the model in place. Otherwise, return a new discretised model. Default is True.
- **check_model** (*bool, optional*) – If True, model checks are performed after discretisation. For large systems these checks can be slow, so can be skipped by setting this option to False. When developing, testing or debugging it is recommened to leave this option as True as it may help to identify any errors. Default is True.

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

Return type *pybamm.BaseModel*

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

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

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

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

Return type tuple

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

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

Returns Discretised symbol

Return type pybamm.expression_tree.symbol.Symbol

set_external_variables(*model*)
Add external variables to the list of variables to account for, being careful about concatenations

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

set_variable_slices(*variables*)
Sets the slicing for variables.

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

1.6.2 Spatial Method

class pybamm.**SpatialMethod**(*options=None*)
A general spatial methods class, with default (trivial) behaviour for some spatial operations. All spatial methods will follow the general form of SpatialMethod in that they contain a method for broadcasting variables onto a mesh, a gradient operator, and a divergence operator.

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

boundary_integral(*child*, *discretised_child*, *region*)
Implements the boundary integral for a spatial method.

Parameters

- **child** (*pybamm.Symbol*) – The symbol to which is being integrated
- **discretised_child** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **region** (*str*) – The region of the boundary over which to integrate. If region is None (default) the integration is carried out over the entire boundary. If region is *negative tab*

or *positive tab* then the integration is only carried out over the appropriate part of the boundary corresponding to the tab.

Returns Contains the result of acting the discretised boundary integral on the child discretised_symbol

Return type class: *pybamm.Array*

boundary_value_or_flux(*symbol*, *discretised_child*, *bcs=None*)

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

Parameters

- **symbol** (*pybamm.Symbol*) – The boundary value or flux symbol
- **discretised_child** (*pybamm.StateVector*) – The discretised variable from which to calculate the boundary value
- **bcs** (*dict (optional)*) – The boundary conditions. If these are supplied and "use bcs" is True in the options, then these will be used to improve the accuracy of the extrapolation.

Returns The variable representing the surface value.

Return type *pybamm.MatrixMultiplication*

broadcast(*symbol*, *domain*, *auxiliary_domains*, *broadcast_type*)

Broadcast symbol to a specified domain.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to be broadcasted
- **domain** (*iterable of strings*) – The domain to broadcast to
- **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 approriate side for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to which is being integrated
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size

divergence(*symbol*, *discretised_symbol*, *boundary_conditions*)

Implements the divergence for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the gradient of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"left": left bc, "right": right bc}})

Returns Contains the result of acting the discretised divergence on the child discretised_symbol

Return type class: *pybamm.Array*

gradient(*symbol*, *discretised_symbol*, *boundary_conditions*)
Implements the gradient for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the gradient of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"left": left bc, "right": right bc}})

Returns Contains the result of acting the discretised gradient on the child discretised_symbol

Return type class: *pybamm.Array*

gradient_squared(*symbol*, *discretised_symbol*, *boundary_conditions*)
Implements the inner product of the gradient with itself for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the gradient of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"left": left bc, "right": right bc}})

Returns Contains the result of taking the inner product of the result of acting the discretised gradient on the child discretised_symbol with itself

Return type class: *pybamm.Array*

indefinite_integral(*child*, *discretised_child*)
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**(*options=None*)

A class which implements the steps specific to the finite volume method during discretisation.

For broadcast and mass_matrix, we follow the default behaviour from SpatialMethod.

Parameters

- **mesh** (*pybamm.Mesh*) – Contains all the submeshes for discretisation
- ****Extends** (“”: *pybamm.SpatialMethod*) –

add_ghost_nodes(*symbol*, *discretised_symbol*, *bcs*)

Add ghost nodes to a symbol.

For Dirichlet bcs, for a boundary condition “y = a at the left-hand boundary”, we concatenate a ghost node to the start of the vector y with value “2*a - y1” where y1 is the value of the first node. Similarly for the right-hand boundary condition.

For Neumann bcs no ghost nodes are added. Instead, the exact value provided by the boundary condition is used at the cell edge when calculating the gradient (see *pybamm.FiniteVolume.add_neumann_values()*).

Parameters

- **symbol** (*pybamm.SpatialVariable*) – The variable to be discretised
- **discretised_symbol** (*pybamm.Vector*) – Contains the discretised variable
- **bcs** (dict of tuples (*pybamm.Scalar*, str)) – Dictionary (with keys “left” and “right”) of boundary conditions. Each boundary condition consists of a value and a flag indicating its type (e.g. “Dirichlet”)

Returns *Matrix @ discretised_symbol + bcs_vector*. When evaluated, this gives the discretised_symbol, with appropriate ghost nodes concatenated at each end.

Return type *pybamm.Symbol*

add_neumann_values(*symbol*, *discretised_gradient*, *bcs*, *domain*)

Add the known values of the gradient from Neumann boundary conditions to the discretised gradient.

Dirichlet bcs are implemented using ghost nodes, see *pybamm.FiniteVolume.add_ghost_nodes()*.

Parameters

- **symbol** (*pybamm.SpatialVariable*) – The variable to be discretised
- **discretised_gradient** (*pybamm.Vector*) – Contains the discretised gradient of symbol
- **bcs** (dict of tuples (*pybamm.Scalar*, str)) – Dictionary (with keys "left" and "right") of boundary conditions. Each boundary condition consists of a value and a flag indicating its type (e.g. "Dirichlet")
- **domain** (*list of strings*) – The domain of the gradient of the symbol (may include ghost nodes)

Returns *Matrix @ discretised_gradient + bcs_vector*. When evaluated, this gives the discretised_gradient, with the values of the Neumann boundary conditions concatenated at each end (if given).

Return type *pybamm.Symbol*

boundary_value_or_flux(*symbol*, *discretised_child*, *bcs=None*)

Uses extrapolation to get the boundary value or flux of a variable in the Finite Volume Method.

See `pybamm.SpatialMethod.boundary_value()`

concatenation(*disc_children*)

Discrete concatenation, taking *edge_to_node* for children that evaluate on edges. See *pybamm.SpatialMethod.concatenation()*

definite_integral_matrix(*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 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*, *method='arithmetic'*)
Convert a discretised symbol evaluated on the cell edges to a discretised symbol evaluated on the cell nodes. See *pybamm.FiniteVolume.shift()*

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

gradient_matrix(*domain*)
Gradient matrix for finite volumes in the appropriate domain. Equivalent to 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,

- *BoundaryValue(F, "left") = 0*, i.e. $3 * F_0 - F_1 = 0$
- $f_{i+1/2} = (F_{i+1} - F_i)/dx_{i+1/2}$

Hence we must have

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

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*, *method='arithmetic'*)
Convert a discretised symbol evaluated on the cell nodes to a discretised symbol evaluated on the cell edges. See *pybamm.FiniteVolume.shift()*

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

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

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

Return type dict

process_binary_operators (*bin_op*, *left*, *right*, *disc_left*, *disc_right*)
Discretise binary operators in model equations. Performs appropriate averaging of diffusivities if one of the children is a gradient operator, so that discretised sizes match up. For this averaging we use the harmonic mean [1].

[1] Recktenwald, Gerald. "The control-volume finite-difference approximation to the diffusion equation." (2012).

Parameters

- **bin_op** (*pybamm.BinaryOperator*) – Binary operator to discretise
- **left** (*pybamm.Symbol*) – The left child of *bin_op*
- **right** (*pybamm.Symbol*) – The right child of *bin_op*
- **disc_left** (*pybamm.Symbol*) – The discretised left child of *bin_op*
- **disc_right** (*pybamm.Symbol*) – The discretised right child of *bin_op*

Returns Discretised binary operator

Return type *pybamm.BinaryOperator*

shift (*discretised_symbol*, *shift_key*, *method*)
Convert a discretised symbol evaluated at edges/nodes, to a discretised symbol evaluated at nodes/edges. Can be the arithmetic mean or the harmonic mean.

Note: when computing fluxes at cell edges it is better to take the harmonic mean based on [1].

[1] Recktenwald, Gerald. "The control-volume finite-difference approximation to the diffusion equation." (2012).

Parameters

- **discretised_symbol** (*pybamm.Symbol*) – Symbol to be averaged. When evaluated, this symbol returns either a scalar or an array of shape (n,) or (n+1,), where n is the number of points in the mesh for the symbol's domain (n = self.mesh[symbol.domain].npts)
- **shift_key** (*str*) – Whether to shift from nodes to edges ("node to edge"), or from edges to nodes ("edge to node")
- **method** (*str*) – Whether to use the "arithmetic" or "harmonic" mean

Returns Averaged symbol. When evaluated, this returns either a scalar or an array of shape (n+1,) (if *shift_key* = *"node to edge"*) or (n,) (if *shift_key* = *"edge to node"*)

Return type *pybamm.Symbol*

spatial_variable(*symbol*)

Creates a discretised spatial variable compatible with the FiniteVolume method.

Parameters **symbol** (*pybamm.SpatialVariable*) – The spatial variable to be discretised.

Returns Contains the discretised spatial variable

Return type *pybamm.Vector*

1.6.4 Scikit Finite Elements

class pybamm.**ScikitFiniteElement**(*options=None*)

A class which implements the steps specific to the finite element method during discretisation. The class uses scikit-fem to discretise the problem to obtain the mass and stiffness matrices. At present, this class is only used for solving the Poisson problem -grad^2 u = f in the y-z plane (i.e. not the through-cell direction).

For broadcast we follow the default behaviour from SpatialMethod.

Parameters

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

assemble_mass_form(*symbol*, *boundary_conditions*, *region='interior'*)

Assembles the form of the finite element mass matrix over the domain interior or boundary.

Parameters

- **symbol** (*pybamm.Variable*) – The variable corresponding to the equation for which we are calculating the mass matrix.
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})
- **region** (*str, optional*) – The domain over which to assemble the mass matrix form. Can be "interior" (default) or "boundary".

Returns The (sparse) mass matrix for the spatial method.

Return type *pybamm.Matrix*

bc_apply(*M*, *boundary*, *zero=False*)

Adjusts the assemled finite element matrices to account for boundary conditons.

Parameters

- **M** (*scipy.sparse.coo_matrix*) – The assemled finite element matrix to adjust.

- **boundary** (numpy.array) – Array of the indicies which correspond to the boundary.
- **zero** (*bool, optional*) – If True, the rows of M given by the indicies in boundary are set to zero. If False, the diagonal element is set to one. default is False.

boundary_integral(*child*, *discretised_child*, *region*)
Implementation of the boundary integral operator. See *pybamm.SpatialMethod.boundary_integral()*

boundary_integral_vector(*domain*, *region*)
A node in the expression tree representing an integral operator over the boundary of a domain

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

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

Parameters

- **domain** (*list*) – The domain(s) of the variable in the integrand
- **region** (*str*) – The region of the boundary over which to integrate. If region is *entire* the integration is carried out over the entire boundary. If region is *negative tab* or *positive tab* then the integration is only carried out over the appropriate part of the boundary corresponding to the tab.

Returns The finite element integral vector for the domain

Return type *pybamm.Matrix*

boundary_mass_matrix(*symbol*, *boundary_conditions*)
Calculates the mass matrix for the finite element method assembled over the boundary.

Parameters

- **symbol** (*pybamm.Variable*) – The variable corresponding to the equation for which we are calculating the mass matrix.
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns The (sparse) mass matrix for the spatial method.

Return type *pybamm.Matrix*

boundary_value_or_flux(*symbol*, *discretised_child*, *bcs=None*)
Returns the average value of the symbol over the negative tab ("negative tab") or the positive tab ("positive tab") in the Finite Element Method.

Overwrites the default pybamm.SpatialMethod.boundary_value()

definite_integral_matrix(*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. The gradient w of the function u is approximated by the finite element method using the same function space as u, i.e. we solve w = grad(u), which corresponds to the weak form w*v*dx = grad(u)*v*dx, where v is a suitable test function.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the laplacian of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns A concatenation that contains the result of acting the discretised gradient on the child discretised_symbol. The first column corresponds to the y-component of the gradient and the second column corresponds to the z component of the gradient.

Return type class: *pybamm.Concatenation*

gradient_matrix (*symbol*, *boundary_conditions*)
Gradient matrix for finite elements in the appropriate domain.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol for which we want to calculate the gradient matrix
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

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

Return type *pybamm.Matrix*

gradient_squared (*symbol*, *discretised_symbol*, *boundary_conditions*)
Multiplication to implement the inner product of the gradient operator with itself. See *pybamm.SpatialMethod.gradient_squared()*

indefinite_integral (*child*, *discretised_child*)
Implementation of the indefinite integral operator. The input discretised child must be defined on the internal mesh edges. See *pybamm.SpatialMethod.indefinite_integral()*

integral (*child*, *discretised_child*)
Vector-vector dot product to implement the integral operator. See *pybamm.SpatialMethod.integral()*

laplacian (*symbol*, *discretised_symbol*, *boundary_conditions*)
Matrix-vector multiplication to implement the laplacian operator.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the laplacian of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size

- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns Contains the result of acting the discretised gradient on the child discretised_symbol

Return type class: *pybamm.Array*

mass_matrix(*symbol*, *boundary_conditions*)
Calculates the mass matrix for the finite element method.

Parameters

- **symbol** (*pybamm.Variable*) – The variable corresponding to the equation for which we are calculating the mass matrix.
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns The (sparse) mass matrix for the spatial method.

Return type *pybamm.Matrix*

spatial_variable(*symbol*)
Creates a discretised spatial variable compatible with the FiniteElement method.

Parameters **symbol** (*pybamm.SpatialVariable*) – The spatial variable to be discretised.

Returns Contains the discretised spatial variable

Return type *pybamm.Vector*

stiffness_matrix(*symbol*, *boundary_conditions*)
Laplacian (stiffness) matrix for finite elements in the appropriate domain.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol for which we want to calculate the laplacian matrix
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol.id: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

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

Return type *pybamm.Matrix*

1.6.5 Zero Dimensional Spatial Method

class pybamm.**ZeroDimensionalMethod**(*options=None*)
A discretisation class for the zero dimensional mesh

Parameters

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

boundary_value_or_flux(*symbol*, *discretised_child*, *bcs=None*)
In 0D, the boundary value is the identity operator. See *SpatialMethod.boundary_value_or_flux()*

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) algebaric 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*, *rtol=1e-06*, *atol=1e-06*, *root_method='casadi'*, *root_tol=1e-06*, *max_steps=1000*)

Solve a discretised model.

Parameters

- **method** (*str, optional*) – The method to use for integration, specific to each solver

- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **root_method** (*str, optional*) – The method to use to find initial conditions (default is "casadi"). If "casadi", the solver uses casadi's Newton rootfinding algorithm to find initial conditions. Otherwise, the solver uses 'scipy.optimize.root' with method specified by 'root_method' (e.g. "lm", "hybr", ...)
- **root_tol** (*float, optional*) – The tolerance for the initial-condition solver (default is 1e-6).
- **max_steps** (*int, optional*) – The maximum number of steps the solver will take before terminating (default is 1000).

calculate_consistent_state(*model*, *time=0*, *y0_guess=None*, *inputs=None*)
Calculate consistent state for the algebraic equations through root-finding

Parameters

- **model** (*pybamm.BaseModel*) – The model for which to calculate initial conditions.
- **time** (*float*) – The time at which to calculate the states
- **y0_guess** (np.array) – Guess for the rootfinding
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving

Returns **y0_consistent** – Initial conditions that are consistent with the algebraic equations (roots of the algebraic equations)

Return type array-like, same shape as y0_guess

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_inputs(*model*, *ext_and_inputs*)
Set values that are controlled externally, such as external variables and input parameters

Parameters **ext_and_inputs** (*dict*) – Any external variables or input parameters to pass to the model when solving

set_up(*model*, *inputs=None*)
Unpack model, perform checks, simplify and calculate jacobian.

Parameters

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes rhs and initial_conditions
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving

solve(*model*, *t_eval*, *external_variables=None*, *inputs=None*)
Execute the solver setup and calculate the solution of the model at specified times.

Parameters

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes rhs and initial_conditions
- **t_eval** (*numeric type*) – The times at which to compute the solution
- **external_variables** (*dict*) – A dictionary of external variables and their corresponding values at the current time
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving

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

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

Step the solution of the model forward by a given time increment. The first time this method is called it executes the necessary setup by calling *self.set_up(model)*.

Parameters

- **old_solution** (*pybamm.Solution* or None) – The previous solution to be added to. If *None*, a new solution is created.
- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes rhs and initial_conditions
- **dt** (*numeric type*) – The timestep over which to step the solution
- **npts** (*int, optional*) – The number of points at which the solution will be returned during the step dt. default is 2 (returns the solution at t0 and t0 + dt).
- **external_variables** (*dict*) – A dictionary of external variables and their corresponding values at the current time
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving
- **save** (*bool*) – Turn on to store the solution of all previous timesteps

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

1.7.3 Scipy Solver

class pybamm.**ScipySolver**(*method='BDF'*, *rtol=1e-06*, *atol=1e-06*)

Solve a discretised model, using scipy._integrate.solve_ivp.

Parameters

- **method** (*str, optional*) – The method to use in solve_ivp (default is "BDF")
- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).

1.7.4 Scikits.odes Solvers

class pybamm.**ScikitsOdeSolver**(*method='cvode'*, *rtol=1e-06*, *atol=1e-06*, *linsolver='dense'*)

Solve a discretised model, using scikits.odes.

Parameters

- **method** (*str, optional*) – The method to use in solve_ivp (default is "BDF")
- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).

- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **linsolver** (*str, optional*) – Can be 'dense' (= default), 'lapackdense', 'spgmr', 'spbcgs', 'sptfqmr'

class pybamm.**ScikitsDaeSolver**(*method='ida', rtol=1e-06, atol=1e-06, root_method='casadi', root_tol=1e-06, max_steps=1000*)

Solve a discretised model, using scikits.odes.

Parameters

- **method** (*str, optional*) – The method to use in solve_ivp (default is "BDF")
- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **root_method** (*str, optional*) – The method to use to find initial conditions (default is "lm")
- **root_tol** (*float, optional*) – The tolerance for the initial-condition solver (default is 1e-6).
- **max_steps** (*int, optional*) – The maximum number of steps the solver will take before terminating (default is 1000).

1.7.5 Casadi Solver

class pybamm.**CasadiSolver**(*mode='safe', rtol=1e-06, atol=1e-06, root_method='casadi', root_tol=1e-06, max_step_decrease_count=5, **extra_options*)

Solve a discretised model, using CasADi.

Extends: *pybamm.BaseSolver*

Parameters

- **method** (*str, optional*) – The method to use for solving the system ('cvodes', for ODEs, or 'idas', for DAEs). Default is 'idas'.
- **mode** (*str*) – How to solve the model (default is "safe"):
  - "fast": perform direct integration, without accounting for events. Recommended when simulating a drive cycle or other simulation where no events should be triggered.
  - "safe": perform step-and-check integration, checking whether events have been triggered. Recommended for simulations of a full charge or discharge.
- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **root_method** (*str, optional*) – The method to use for finding consistend initial conditions. Default is 'lm'.
- **root_tol** (*float, optional*) – The tolerance for root-finding. Default is 1e-6.
- **max_step_decrease_counts** (*float, optional*) – The maximum number of times step size can be decreased before an error is raised. Default is 5.
- **extra_options** (*keyword arguments, optional*) – Any extra keyword-arguments; these are passed directly to the CasADi integrator. Please consult CasADi documentation for details.

1.7.6 Solution

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

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

Parameters

- **t** (numpy.array, size (n,)) – A one-dimensional array containing the times at which the solution is evaluated
- **y** (numpy.array, size (m, n)) – A two-dimensional array containing the values of the solution. y[i, :] is the vector of solutions at time t[i].
- **t_event** (numpy.array, size (1,)) – A zero-dimensional array containing the time at which the event happens.
- **y_event** (numpy.array, size (m,)) – A one-dimensional array containing the value of the solution at the time when the event happens.
- **termination** (*str*) – String to indicate why the solution terminated
- **copy_this** (*pybamm.Solution*, optional) – A solution to copy, if provided. Default is None.

inputs
Values of the inputs

model
Model used for solution

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

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

Parameters

- **filename** (*str*) – The name of the file to save data to
- **variables** (*list, optional*) – List of variables to save. If None, saves all of the variables that have been created so far
- **to_format** (*str, optional*) – The format to save to. Options are:
  - 'pickle' (default): creates a pickle file with the data dictionary
  - 'matlab': creates a .mat file, for loading in matlab
  - 'csv': creates a csv file (1D variables only)

t
Times at which the solution is evaluated

t_event
Time at which the event happens

termination
Reason for termination

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

y
Values of the solution

y_event
Value of the solution at the time of the event

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

Extends: *_BaseSolution*

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

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

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

1.8 Experiments

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

1.8.1 Base Experiment Class

class pybamm.**Experiment**(*operating_conditions*, *parameters=None*, *period='1 minute'*)
Base class for experimental conditions under which to run the model. In general, a list of operating conditions should be passed in. Each operating condition should be of the form "Do this for this long" or "Do this until this happens". For example, "Charge at 1 C for 1 hour", or "Charge at 1 C until 4.2 V", or "Charge at 1 C for 1 hour or until 4.2 V". The instructions can be of the form "(Dis)charge at x A/C/W", "Rest", or "Hold at x V". The running time should be a time in seconds, minutes or hours, e.g. "10 seconds", "3 minutes" or "1 hour". The stopping conditions should be a circuit state, e.g. "1 A", "C/50" or "3 V".

Parameters

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

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

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

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

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

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

Return type list

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

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

1.9 Post-Process Variables

class pybamm.**ProcessedVariable**(*base_variable*, *solution*, *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)
- **solution** (*pybamm.Solution*) – The solution object to be used to create the processed variables
- **interp_kind** (*str*) – The method to use for interpolation
- **known_evals** (*dict*) – Dictionary of known evaluations, to be used to speed up finding the solution

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

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

data
Same as entries, but different name

initialise_3D()
Initialise a 3D object that depends on x and r, or x and z.

1.10 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
```

(continues on next page)

(continued from previous page)

```
>>> 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.

1.11 Simulation

class pybamm.**Simulation**(*model*, *experiment=None*, *geometry=None*, *parameter_values=None*, *submesh_types=None*, *var_pts=None*, *spatial_methods=None*, *solver=None*, *quick_plot_vars=None*, *C_rate=None*)

A Simulation class for easy building and running of PyBaMM simulations.

Parameters

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

- **geometry** (*pybamm.Geometry* (optional)) – The geometry upon which to solve the model
- **parameter_values** (*dict (optional)*) – A dictionary of parameters and their corresponding numerical values
- **submesh_types** (*dict (optional)*) – A dictionary of the types of submesh to use on each subdomain
- **var_pts** (*dict (optional)*) – A dictionary of the number of points used by each spatial variable
- **spatial_methods** (*dict (optional)*) – A dictionary of the types of spatial method to use on each domain (e.g. pybamm.FiniteVolume)
- **solver** (*pybamm.BaseSolver* (optional)) – The solver to use to solve the model.
- **quick_plot_vars** (*list (optional)*) – A list of variables to plot automatically
- **C_rate** (*float (optional)*) – The C_rate at which you would like to run a constant current experiment at.

build(*check_model=True*)

A method to build the model into a system of matrices and vectors suitable for performing numerical computations. If the model has already been built or solved then this function will have no effect. If you want to rebuild, first use "reset()". This method will automatically set the parameters if they have not already been set.

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

get_variable_array(**variables*)

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

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

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

Return type dict

plot(*quick_plot_vars=None*, *testing=False*)

A method to quickly plot the outputs of the simulation.

Parameters

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

reset(*update_model=True*)

A method to reset a simulation back to its unprocessed state.

save(*filename*)

Save simulation using pickle

set_defaults()

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

set_parameters()

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

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

solve(*t_eval=None*, *solver=None*, *external_variables=None*, *inputs=None*, *check_model=True*)
A method to solve the model. This method will automatically build and set the model parameters if not already done so.

Parameters

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

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

Parameters

- **model_options** (*dict, optional*) – A dictionary of options to tweak the model you are using
- **geometry** (*pybamm.Geometry*, optional) – The geometry upon which to solve the model
- **parameter_values** (*dict, optional*) – A dictionary of parameters and their corresponding numerical values
- **submesh_types** (*dict, optional*) – A dictionary of the types of submesh to use on each subdomain
- **var_pts** (*dict, optional*) – A dictionary of the number of points used by each spatial variable
- **spatial_methods** (*dict, optional*) – A dictionary of the types of spatial method to use on each domain (e.g. pybamm.FiniteVolume)
- **solver** (*pybamm.BaseSolver* (optional)) – The solver to use to solve the model.
- **quick_plot_vars** (*list (optional)*) – A list of variables to plot automatically
- **C_rate** (*float (optional)*) – The C_rate at which you would like to run a constant current experiment at.

step(*dt*, *solver=None*, *npts=2*, *external_variables=None*, *inputs=None*, *save=True*)
A method to step the model forward one timestep. This method will automatically build and set the model parameters if not already done so.

Parameters

- **dt** (*numeric type*) – The timestep over which to step the solution
- **solver** (*pybamm.BaseSolver*) – The solver to use to solve the model.
- **npts** (*int, optional*) – The number of points at which the solution will be returned during the step dt. default is 2 (returns the solution at t0 and t0 + dt).
- **external_variables** (*dict*) – A dictionary of external variables and their corresponding values at the current time. The variables must correspond to the variables that would normally be found by solving the submodels that have been made external.
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving
- **save** (*bool*) – Turn on to store the solution of all previous timesteps

1.12 Citations

class pybamm.**Citations**

Entry point to citations management. This object may be used to record Bibtex citation information and then register that a particular citation is relevant for a particular simulation. For a list of all possible citations, see *pybamm/CITATIONS.txt*

Examples

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

print(*filename=None*)

Print all citations that were used for running simulations.

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

read_citations()

Read the citations text file

register(*key*)

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

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

pybamm.**print_citations**(*filename=None*)

See *Citations.print()*

1.13 Parameters command line interface

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

copy a given parameter directory to the package directory. To get a list of currently available parameter sets, use `pybamm_list_parameters`.

`pybamm.parameters_cli.`**`add_parameter`**(*arguments=None*)

Add a parameter directory to package input directory. This allows the parameters to be used from anywhere in the system.

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

`pybamm.parameters_cli.`**`edit_parameter`**(*arguments=None*)

Copy a given parameter package directory to the current working directory for editing. The copy preserves the directory structure within the "input" directory, i.e

```
edit_param(["graphite_Kim2011","lithium-ion","anodes"])
```

will create the directory structure "input/parameters/lithium-ion/anodes/graphite_Kim2011" in the current working directory.

`pybamm.parameters_cli.`**`list_parameters`**(*arguments=None*)

Output a list of available parameter sets for a given chemistry and component. The list is divided into package parameter serts and local parameter sets, located in the current working directory.

```
>>> from pybamm.parameters_cli import list_parameters
>>> list_parameters(["lithium-ion", "anodes"])
Available package parameters:
  * graphite_Chen2020
  * graphite_mcmb2528_Marquis2019
  * graphite_Kim2011
Available local parameters:
```

CHAPTER 2

Examples

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

CHAPTER 3

Contributing

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

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

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

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

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

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

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/lithium_ion/anodes/new_anode_chemistry_AuthorYear/diffusivity_AuthorYear.py`

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

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

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

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

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

3.1.4 Adding data for interpolation

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

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

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

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

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

3.1.5 Using new parameters

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

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

Then, to use these new parameters, use:

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

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

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

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

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

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

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

3.1.6 Unit tests for the new class

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

3.1.7 Test on the models

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

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

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

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

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

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

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

is simply written as

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

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

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 if zero and then write the right hand side into the `self.algebraic` dictionary in the same way. The resulting dictionary should look like:

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

Boundary conditions

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

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

Initial conditions

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

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

Output variables

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

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

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

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

Events

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

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

Events will stop the solver whenever they return 0.

Setting defaults

It can be useful for testing, and quickly running a model to have a default setup. Each of the defaults listed above should adhere to the API requirements but in short, we require `self.default_geometry` to be 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 –> Addition), and calls upon spatial methods to perform the harder conversions (e.g. grad(u) –> Matrix * StateVector, SpatialVariable –> Vector, etc).

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

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

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*, which sets up some useful solver properties such as tolerances and implement a method `self.solve()` that solves a model at a given set of time points.

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

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

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 *pybamm.BaseSolver*. For example:

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

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

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

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

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

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.

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

T

U

V

X

Y

Z