

---

# **cbcbeat Documentation**

*Release 1.0*

**cbcbeat-authors**

**Nov 29, 2018**



---

## Contents

---

<b>1</b>	<b>Installation and dependencies:</b>	<b>3</b>
<b>2</b>	<b>Main authors:</b>	<b>5</b>
<b>3</b>	<b>License:</b>	<b>7</b>
<b>4</b>	<b>Testing and verification:</b>	<b>9</b>
<b>5</b>	<b>Contributions:</b>	<b>11</b>
<b>6</b>	<b>Documentation:</b>	<b>13</b>
6.1	Examples and demos: . . . . .	13
6.2	API documentation: . . . . .	13
<b>7</b>	<b>Indices and tables</b>	<b>37</b>
	<b>Python Module Index</b>	<b>39</b>



cbcbeat is a Python-based software collection targeting computational cardiac electrophysiology problems. cbcbeat contains solvers of varying complexity and performance for the classical monodomain and bidomain equations coupled with cardiac cell models. The cbcbeat solvers are based on algorithms described in Sundnes et al (2006) and the core FEniCS Project software (Logg et al, 2012). All cbcbeat solvers allow for automated derivation and computation of adjoint and tangent linear solutions, functional derivatives and Hessians via the dolfin-adjoint software (Farrell et al, 2013). The computation of functional derivatives in turn allows for automated and efficient solution of optimization problems such as those encountered in data assimilation or other inverse problems.

cbcbeat is based on the finite element functionality provided by the FEniCS Project software, the automated derivation and computation of adjoints offered by the dolfin-adjoint software and cardiac cell models from the CellML repository.

cbcbeat originates from the [Center for Biomedical Computing](#), a Norwegian Centre of Excellence, hosted by [Simula Research Laboratory](#), Oslo, Norway.



# CHAPTER 1

---

## Installation and dependencies:

---

The cbcbeat source code is hosted on Bitbucket:

<https://bitbucket.org/meg/cbcbeat>

The cbcbeat solvers are based on the [FEniCS Project](#) finite element library and its extension [dolfin-adjoint](#). Any type of build of FEniCS and dolfin-adjoint should work, but cbcbeat has mainly been developed using native source builds and is mainly tested via Docker images.

See the separate file `INSTALL` in the root directory of the cbcbeat source for a complete list of dependencies and installation instructions.





## CHAPTER 2

---

Main authors:

---

See the separate file `AUTHORS` in the root directory of the `cbcbeat` source for the list of authors and contributors.



## CHAPTER 3

---

License:

---

cbcbeat is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

cbcbeat is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with cbcbeat. If not, see <<http://www.gnu.org/licenses/>>.



## CHAPTER 4

---

### Testing and verification:

---

The cbcbeat test suite is based on [pytest](#) and available in the `test/` directory of the cbcbeat source. See the `INSTALL` file in the root directory of the cbcbeat source for how to run the tests.

cbcbeat uses Bitbucket Pipelines for automated and continuous testing, see the current test status of cbcbeat here:

<https://bitbucket.org/meg/cbcbeat/addon/pipelines/home>



## CHAPTER 5

---

### Contributions:

---

Contributions to cbcbat are very welcome. If you are interested in improving or extending the software please [contact us](#) via the issues or pull requests on Bitbucket. Similarly, please [report](#) issues via Bitbucket.





The cbcbeat solvers are based on the Python interface of the [FEniCS Project](#) finite element library and its extension [dolfin-adjoint](#). We recommend users of cbcbeat to first familiarize with these libraries. The [FEniCS tutorial](#) and the [dolfin-adjoint documentation](#) are good starting points for new users.

## 6.1 Examples and demos:

A collection of examples on how to use cbcbeat is available in the `demo/` directory of the cbcbeat source. We also recommend looking at the test suite for examples of how to use the cbcbeat solvers.

## 6.2 API documentation:

### 6.2.1 cbcbeat package

#### Subpackages

#### `cbcbeat.cellmodels` package

#### Submodules

#### `cbcbeat.cellmodels.beeler_reuter_1977` module

This module contains a `Beeler_reuter_1977` cardiac cell model

The module was autogenerated from a gotran ode file

```
class cbcbeat.cellmodels.beeler_reuter_1977.Beeler_reuter_1977 (params=None,  
                                                    init_conditions=None)
```

```
    Bases: cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel
```

**F** (*v, s, time=None*)  
 Right hand side for ODE system

**I** (*v, s, time=None*)  
 Transmembrane current

$$I = -dV/dt$$

**static default\_initial\_conditions** ()  
 Set-up and return default initial conditions.

**static default\_parameters** ()  
 Set-up and return default parameters.

**num\_states** ()

### cbcbeat.cellmodels.cardiaccellmodel module

This module contains a base class for cardiac cell models.

**class** cbcbeat.cellmodels.cardiaccellmodel.**CardiacCellModel** (*params=None, init\_conditions=None*)

Base class for cardiac cell models. Specialized cell models should subclass this class.

Essentially, a cell model represents a system of ordinary differential equations. A cell model is here described by two (Python) functions, named F and I. The model describes the behaviour of the transmembrane potential 'v' and a number of state variables 's'

The function F gives the right-hand side for the evolution of the state variables:

$$d/dt s = F(v, s)$$

The function I gives the ionic current. If a single cell is considered, I gives the (negative) right-hand side for the evolution of the transmembrane potential

$$(*) d/dt v = - I(v, s)$$

If used in a bidomain setting, the ionic current I enters into the parabolic partial differential equation of the bidomain equations.

If a stimulus is provided via

```
cell = CardiacCellModel() cell.stimulus = Expression("I_s(t)", degree=1)
```

then I\_s is added to the right-hand side of (\*), which thus reads

$$d/dt v = - I(v, s) + I_s$$

Note that the cardiac cell model stimulus is ignored when the cell model is used a spatially-varying setting (for instance in the bidomain setting). In this case, the user is expected to specify a stimulus for the cardiac model instead.

**F** (*v, s, time=None*)  
 Return right-hand side for state variable evolution.

**I** (*v, s, time=None*)  
 Return the ionic current.

**static default\_initial\_conditions** ()  
 Set-up and return default initial conditions.

**static default\_parameters** ()  
 Set-up and return default parameters.

**initial\_conditions** ()  
Return initial conditions for v and s as an Expression.

**num\_states** ()  
Return number of state variables (in addition to the membrane potential).

**parameters** ()  
Return the current parameters.

**set\_initial\_conditions** (\*\*init)  
Update initial\_conditions in model

**set\_parameters** (\*\*params)  
Update parameters in model

**class** cbcbeat.cellmodels.cardiaccellmodel.**MultiCellModel** (models, keys, markers)

Bases: *cbcbeat.cellmodels.cardiaccellmodel.Cardiaccellmodel*

**F** (v, s, time=None, index=None)

**I** (v, s, time=None, index=None)

**initial\_conditions** ()  
Return initial conditions for v and s as a dolfin.GenericFunction.

**keys** ()

**markers** ()

**mesh** ()

**models** ()

**num\_models** ()

**num\_states** ()  
Return number of state variables (in addition to the membrane potential).

### cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered module

This module contains a Fenton\_karma\_1998\_BR\_altered cardiac cell model

The module was autogenerated from a gotran ode file

**class** cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered.**Fenton\_karma\_1998\_BR\_altered** (params=None, init\_conditions=None)

Bases: *cbcbeat.cellmodels.cardiaccellmodel.Cardiaccellmodel*

**F** (v, s, time=None)  
Right hand side for ODE system

**I** (v, s, time=None)  
Transmembrane current

$$I = -dV/dt$$

**static default\_initial\_conditions** ()  
Set-up and return default initial conditions.

**static default\_parameters** ()  
Set-up and return default parameters.

**num\_states** ()

### cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered module

This module contains a Fenton\_karma\_1998\_MLR-1\_altered cardiac cell model

The module was autogenerated from a gotran ode file

```
class cbcbeat.cellmodels.fenton_karma_1998_MLR1_altered.Fenton_karma_1998_MLR1_altered (params=None, init_c)
    Bases: cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel
    F (v, s, time=None)
        Right hand side for ODE system
    I (v, s, time=None)
        Transmembrane current
        I = -dV/dt
    static default_initial_conditions ()
        Set-up and return default initial conditions.
    static default_parameters ()
        Set-up and return default parameters.
    num_states ()
```

### cbcbeat.cellmodels.fitzhughnagumo module

This module contains a Fitzhughnagumo cardiac cell model

The module was autogenerated from a gotran form file

```
class cbcbeat.cellmodels.fitzhughnagumo.Fitzhughnagumo (params=None, init_conditions=None)
    Bases: cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel
    NOT_IMPLEMENTED
    F (v, s, time=None)
        Right hand side for ODE system
    I (v, s, time=None)
        Transmembrane current
    static default_initial_conditions ()
        Set-up and return default initial conditions.
    static default_parameters ()
        Set-up and return default parameters.
    num_states ()
```

### cbcbeat.cellmodels.fitzhughnagumo\_manual module

This module contains a FitzHugh-Nagumo cardiac cell model

The module was written by hand, in particular it was not autogenerated.

```
class cbcbeat.cellmodels.fitzhughnagumo_manual.FitzHughNagumoManual (params=None, init_conditions=None)
    Bases: cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel
```

A reparametrized FitzHughNagumo model, based on Section 2.4.1 in “Computing the electrical activity in the heart” by Sundnes et al, 2006.

This is a model containing two nonlinear, ODEs for the evolution of the transmembrane potential  $v$  and one additional state variable  $s$ .

```
F ( $v, s, time=None$ )
    Return right-hand side for state variable evolution.

I ( $v, s, time=None$ )
    Return the ionic current.

static default_initial_conditions ()

static default_parameters ()
    Set-up and return default parameters.

num_states ()
    Return number of state variables.
```

### cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010 module

This module contains a Grandi\_pasqualini\_bers\_2010 cardiac cell model

The module was autogenerated from a gotran ode file

```
class cbcbeat.cellmodels.grandi_pasqualini_bers_2010.Grandi_pasqualini_bers_2010 ( $params=None$ ,
                                                                     $init_conditions=None$ )

    Bases: cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel

    F ( $v, s, time=None$ )
        Right hand side for ODE system

    I ( $v, s, time=None$ )
        Transmembrane current

         $I = -dV/dt$ 

    static default_initial_conditions ()
        Set-up and return default initial conditions.

    static default_parameters ()
        Set-up and return default parameters.

    num_states ()
```

### cbcbeat.cellmodels.nocellmodel module

This module contains a dummy cardiac cell model.

```
class cbcbeat.cellmodels.nocellmodel.NoCellModel ( $params=None$ ,
                                                                     $init_conditions=None$ )

    Bases: cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel

    Class representing no cell model (only bidomain equations). It actually just represents a single completely decoupled ODE.

    F ( $v, s, time=None$ )

    I ( $v, s, time=None$ )
```

```

static default_initial_conditions ()
    Set-up and return default initial conditions.

num_states ()

```

### cbcbeat.cellmodels.rogers\_mcculloch\_manual module

This module contains a Rogers-McCulloch cardiac cell model which is a modified version of the FitzHughNagumo model.

This formulation is based on the description on page 2 of “Optimal control approach ...” by Nagaiah, Kunisch and Plank, 2013, J Math Biol.

The module was written by hand, in particular it was not autogenerated.

```

class cbcbeat.cellmodels.rogers_mcculloch_manual.RogersMcCulloch (params=None,
                                                                init_conditions=None)

```

Bases: *cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel*

**The Rogers-McCulloch model is a modified FitzHughNagumo model.** This formulation follows the description on page 2 of “Optimal control approach ...” by Nagaiah, Kunisch and Plank, 2013, J Math Biol with  $w$  replaced by  $s$ . Note that this model introduces one additional parameter compared to the original 1994 Rogers-McCulloch model.

This is a model containing two nonlinear, ODEs for the evolution of the transmembrane potential  $v$  and one additional state variable  $s$ :

$$\text{rac}\{dv\}\{dt\} = -I_{ion}(v, s)$$

$$\text{rac}\{ds\}\{dt\} = F(v, s)$$

where

$$I_{ion}(v, s) = gv(1 - v/v_{th})(1 - v/v_p) + \eta_1vs$$

$$F(v, s) = \eta_2(v/v_p - \eta_3s)$$

```

F (v, s, time=None)
    Return right-hand side for state variable evolution.

```

```

I (v, s, time=None)
    Return the ionic current.

```

```

static default_initial_conditions ()

```

```

static default_parameters ()
    Set-up and return default parameters.

```

```

num_states ()
    Return number of state variables.

```

### cbcbeat.cellmodels.tentusscher\_2004\_mcell module

This module contains a Tentusscher\_2004\_mcell cardiac cell model

The module was autogenerated from a gotran ode file

```

class cbcbeat.cellmodels.tentusscher_2004_mcell.Tentusscher_2004_mcell (params=None,
                                                                init_conditions=None)

```

Bases: *cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel*

NOT\_IMPLEMENTED

**F** (*v, s, time=None*)  
 Right hand side for ODE system

**I** (*v, s, time=None*)  
 Transmembrane current

$I = -dV/dt$

**static default\_initial\_conditions** ()  
 Set-up and return default initial conditions.

**static default\_parameters** ()  
 Set-up and return default parameters.

**num\_states** ()

### cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont module

This module contains a Tentusscher\_2004\_mcell\_cont cardiac cell model

The module was autogenerated from a gotran ode file

**class** cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont.**Tentusscher\_2004\_mcell\_cont** (*params=None, init\_conditions=None*)

Bases: *cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel*

**F** (*v, s, time=None*)  
 Right hand side for ODE system

**I** (*v, s, time=None*)  
 Transmembrane current

$I = -dV/dt$

**static default\_initial\_conditions** ()  
 Set-up and return default initial conditions.

**static default\_parameters** ()  
 Set-up and return default parameters.

**num\_states** ()

### cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc module

This module contains a Tentusscher\_2004\_mcell\_disc cardiac cell model

The module was autogenerated from a gotran ode file

**class** cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc.**Tentusscher\_2004\_mcell\_disc** (*params=None, init\_conditions=None*)

Bases: *cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel*

**F** (*v, s, time=None*)  
 Right hand side for ODE system

**I** (*v, s, time=None*)  
 Transmembrane current

$I = -dV/dt$

**static default\_initial\_conditions** ()  
 Set-up and return default initial conditions.

```
static default_parameters ()  
    Set-up and return default parameters.  
  
num_states ()
```

### cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell module

This module contains a Tentusscher\_panfilov\_2006\_M\_cell cardiac cell model

The module was autogenerated from a gotran ode file

```
class cbcbeat.cellmodels.tentusscher_panfilov_2006_M_cell.Tentusscher_panfilov_2006_M_cell
```

```
    Bases: cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel
```

```
F (v, s, time=None)  
    Right hand side for ODE system
```

```
I (v, s, time=None)  
    Transmembrane current  
  
     $I = -dV/dt$ 
```

```
static default_initial_conditions ()  
    Set-up and return default initial conditions.
```

```
static default_parameters ()  
    Set-up and return default parameters.
```

```
num_states ()
```

### cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell module

This module contains a Tentusscher\_panfilov\_2006\_epi\_cell cardiac cell model

The module was autogenerated from a gotran ode file

```
class cbcbeat.cellmodels.tentusscher_panfilov_2006_epi_cell.Tentusscher_panfilov_2006_epi_c
```

```
    Bases: cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel
```

```
F (v, s, time=None)  
    Right hand side for ODE system
```

```
I (v, s, time=None)  
    Transmembrane current  
  
     $I = -dV/dt$ 
```

```
static default_initial_conditions ()  
    Set-up and return default initial conditions.
```

```
static default_parameters ()  
    Set-up and return default parameters.
```

```
num_states ()
```



## Module contents

### Submodules

#### cbcbeat.bidomainsolver module

These solvers solve the (pure) bidomain equations on the form: find the transmembrane potential  $v = v(x, t)$  and the extracellular potential  $u = u(x, t)$  such that

$$\begin{aligned} v_t - \operatorname{div}(G_i v + G_i u) &= I_s \\ \operatorname{div}(G_i v + (G_i + G_e)u) &= I_a \end{aligned}$$

where the subscript  $t$  denotes the time derivative;  $G_x$  denotes a weighted gradient:  $G_x = M_x \operatorname{grad}(v)$  for  $x \in \{i, e\}$ , where  $M_i$  and  $M_e$  are the intracellular and extracellular cardiac conductivity tensors, respectively;  $I_s$  and  $I_a$  are prescribed input. In addition, initial conditions are given for  $v$ :

$$v(x, 0) = v_0$$

Finally, boundary conditions must be prescribed. For now, this solver assumes pure homogeneous Neumann boundary conditions for  $v$  and  $u$  and enforces the additional average value zero constraint for  $u$ .

```
class cbcbeat.bidomainsolver.BasicBidomainSolver(mesh, time, M_i, M_e,
                                                I_s=None, I_a=None, v_=None,
                                                params=None)
```

Bases: object

This solver is based on a theta-scheme discretization in time and CG\_1 x CG\_1 (x R) elements in space.

---

**Note:** For the sake of simplicity and consistency with other solver objects, this solver operates on its solution fields (as state variables) directly internally. More precisely, solve (and step) calls will act by updating the internal solution fields. It implies that initial conditions can be set (and are intended to be set) by modifying the solution fields prior to simulation.

---

#### Arguments

**mesh** (**dolfin.Mesh**) The spatial domain (mesh)

**time** (**dolfin.Constant** or **None**) A constant holding the current time. If None is given, time is created for you, initialized to zero.

**M\_i** (**ufl.Expr**) The intracellular conductivity tensor (as an UFL expression)

**M\_e** (**ufl.Expr**) The extracellular conductivity tensor (as an UFL expression)

**I\_s** (**dict**, **optional**) A typically time-dependent external stimulus given as a dict, with domain markers as the key and a **dolfin.Expression** as values. NB: it is assumed that the time dependence of **I\_s** is encoded via the 'time' Constant.

**I\_a** (**dolfin.Expression**, **optional**) A (typically time-dependent) external applied current

**v\_** (**ufl.Expr**, **optional**) Initial condition for  $v$ . A new **dolfin.Function** will be created if none is given.

**params** (**dolfin.Parameters**, **optional**) Solver parameters

**static default\_parameters** ()

Initialize and return a set of default parameters

**Returns** A set of parameters (`dolfin.Parameters`)

To inspect all the default parameters, do:

```
info(BasicBidomainSolver.default_parameters(), True)
```

**solution\_fields()**

Return tuple of previous and current solution objects.

Modifying these will modify the solution objects of the solver and thus provides a way for setting initial conditions for instance.

**Returns** (previous `v`, current `vur`) (tuple of `dolfin.Function`)

**solve** (*interval*, *dt=None*)

Solve the discretization on a given time interval (`t0`, `t1`) with a given timestep `dt` and return generator for a tuple of the interval and the current solution.

**Arguments**

**interval (tuple)** The time interval for the solve given by (`t0`, `t1`)

**dt (int, optional)** The timestep for the solve. Defaults to length of interval

**Returns** (timestep, `solution_fields`) via (`genexpr`)

*Example of usage:*

```
# Create generator
solutions = solver.solve((0.0, 1.0), 0.1)

# Iterate over generator (computes solutions as you go)
for (interval, solution_fields) in solutions:
    (t0, t1) = interval
    v_, vur = solution_fields
    # do something with the solutions
```

**step** (*interval*)

Solve on the given time interval (`t0`, `t1`).

**Arguments**

**interval (tuple)** The time interval (`t0`, `t1`) for the step

**Invariants** Assuming that `v_` is in the correct state for `t0`, gives `self.vur` in correct state at `t1`.

**time**

The internal time of the solver.

**class** `cbcbeat.bidomainsolver.BidomainSolver` (*mesh*, *time*, *M\_i*, *M\_e*, *I\_s=None*, *I\_a=None*, *v\_=None*, *params=None*)

Bases: `cbcbeat.bidomainsolver.BasicBidomainSolver`

This solver is based on a theta-scheme discretization in time and `CG_1` x `CG_1` (x `R`) elements in space.

---

**Note:** For the sake of simplicity and consistency with other solver objects, this solver operates on its solution fields (as state variables) directly internally. More precisely, `solve` (and `step`) calls will act by updating the internal solution fields. It implies that initial conditions can be set (and are intended to be set) by modifying the solution fields prior to simulation.

---

**Arguments**

**mesh** (`dolfin.Mesh`) The spatial domain (mesh)

**time** (`dolfin.Constant` or `None`) A constant holding the current time. If `None` is given, time is created for you, initialized to zero.

**M<sub>i</sub>** (`ufl.Expr`) The intracellular conductivity tensor (as an UFL expression)

**M<sub>e</sub>** (`ufl.Expr`) The extracellular conductivity tensor (as an UFL expression)

**I<sub>s</sub>** (`dict`, `optional`) A typically time-dependent external stimulus given as a dict, with domain markers as the key and a `dolfin.Expression` as values. NB: it is assumed that the time dependence of `Is` is encoded via the 'time' Constant.

**I<sub>a</sub>** (`dolfin.Expression`, `optional`) A (typically time-dependent) external applied current

**v<sub>-</sub>** (`ufl.Expr`, `optional`) Initial condition for `v`. A new `dolfin.Function` will be created if none is given.

**params** (`dolfin.Parameters`, `optional`) Solver parameters

**static default\_parameters** ()

Initialize and return a set of default parameters

**Returns** A set of parameters (`dolfin.Parameters`)

To inspect all the default parameters, do:

```
info(BidomainSolver.default_parameters(), True)
```

**linear\_solver**

The linear solver (`dolfin.LUSolver` or `dolfin.PETScKrylovSolver`).

**nullspace**

**step** (*interval*)

Solve on the given time step (`t0`, `t1`).

**Arguments**

**interval** (`tuple`) The time interval (`t0`, `t1`) for the step

**Invariants** Assuming that `v-` is in the correct state for `t0`, gives `self.vur` in correct state at `t1`.

**variational\_forms** (*k<sub>n</sub>*)

Create the variational forms corresponding to the given discretization of the given system of equations.

**Arguments**

**k<sub>n</sub>** (`ufl.Expr` or `float`) The time step

**Returns** (`lhs`, `rhs`) (`tuple` of `ufl.Form`)

## cbcbeat.cardiacmodels module

This module contains a container class for cardiac models: `CardiacModel`. This class should be instantiated for setting up specific cardiac simulation scenarios.

**class** `cbcbeat.cardiacmodels.CardiacModel` (*domain*, *time*, *M<sub>i</sub>*, *M<sub>e</sub>*, *cell\_models*, *stimulus=None*, *applied\_current=None*)

Bases: `object`

A container class for cardiac models. Objects of this class represent a specific cardiac simulation set-up and should provide

- A computational domain
- A cardiac cell model
- Intra-cellular and extra-cellular conductivities
- Various forms of stimulus (optional).

This container class is designed for use with the splitting solvers (*cbcbeat.splittingsolver*), see their documentation for more information on how the attributes are interpreted in that context.

### *Arguments*

**domain** (*dolfin.Mesh*) the computational domain in space

**time** (*dolfin.Constant* or *None*) A constant holding the current time.

**M<sub>i</sub>** (*ufl.Expr*) the intra-cellular conductivity as an ufl Expression

**M<sub>e</sub>** (*ufl.Expr*) the extra-cellular conductivity as an ufl Expression

**cell\_models** (*CardiacCellModel*) a cell model or a dict with cell models associated with a cell model domain

**stimulus** (*dict*, *optional*) A typically time-dependent external stimulus given as a dict, with domain markers as the key and a *dolfin.Expression* as values. NB: it is assumed that the time dependence of I<sub>s</sub> is encoded via the 'time' Constant.

**applied\_current** (*ufl.Expr*, *optional*) an applied current as an ufl Expression

**applied\_current** ()

An applied current: used as a source in the elliptic bidomain equation

**cell\_models** ()

Return the cell models

**conductivities** ()

Return the intracellular and extracellular conductivities as a tuple of UFL Expressions.

*Returns* (M<sub>i</sub>, M<sub>e</sub>) (tuple of *ufl.Expr*)

**domain** ()

The spatial domain (*dolfin.Mesh*).

**extracellular\_conductivity** ()

The intracellular conductivity (*ufl.Expr*).

**intracellular\_conductivity** ()

The intracellular conductivity (*ufl.Expr*).

**stimulus** ()

A stimulus: used as a source in the parabolic bidomain equation

**time** ()

The current time (*dolfin.Constant* or *None*).

### **cbcbeat.cellsolver module**

This module contains solvers for (subclasses of) *CardiacCellModel*.

**class** *cbcbeat.cellsolver.BasicSingleCellSolver* (*model*, *time*, *params=None*)

Bases: *cbcbeat.cellsolver.BasicCardiacODESolver*

A basic, non-optimised solver for systems of ODEs typically encountered in cardiac applications of the form: find a scalar field  $v = v(t)$  and a vector field  $s = s(t)$

$$\begin{aligned} v_t &= -I_{ion}(v, s) + I_s \\ s_t &= F(v, s) \end{aligned}$$

where  $I_{ion}$  and  $F$  are given non-linear functions,  $I_s$  is some prescribed stimulus. If  $I_s$  depends on time, it is assumed that  $I_s$  is a `dolfin.Expression` with parameter 't'.

Use this solver if you just want to test the results from a cardiac cell model without any spatial mesh dependence.

Here, this nonlinear ODE system is solved via a theta-scheme. By default `theta=0.5`, which corresponds to a Crank-Nicolson scheme. This can be changed by modifying the solver parameters.

---

**Note:** For the sake of simplicity and consistency with other solver objects, this solver operates on its solution fields (as state variables) directly internally. More precisely, `solve` (and `step`) calls will act by updating the internal solution fields. It implies that initial conditions can be set (and are intended to be set) by modifying the solution fields prior to simulation.

---

### Arguments

- model** (`CardiacCellModel`) A cardiac cell model
- time** (`Constant` or `None`) A constant holding the current time.
- params** (`dolfin.Parameters`, optional) Solver parameters

**class** `cbcbeat.cellsolver.BasicCardiacODESolver` (*mesh*, *time*, *model*, *I\_s=None*,  
*params=None*)

Bases: `object`

A basic, non-optimised solver for systems of ODEs typically encountered in cardiac applications of the form: find a scalar field  $v = v(x, t)$  and a vector field  $s = s(x, t)$

$$\begin{aligned} v_t &= -I_{ion}(v, s) + I_s \\ s_t &= F(v, s) \end{aligned}$$

where  $I_{ion}$  and  $F$  are given non-linear functions, and  $I_s$  is some prescribed stimulus.

Here, this nonlinear ODE system is solved via a theta-scheme. By default `theta=0.5`, which corresponds to a Crank-Nicolson scheme. This can be changed by modifying the solver parameters.

---

**Note:** For the sake of simplicity and consistency with other solver objects, this solver operates on its solution fields (as state variables) directly internally. More precisely, `solve` (and `step`) calls will act by updating the internal solution fields. It implies that initial conditions can be set (and are intended to be set) by modifying the solution fields prior to simulation.

---

### Arguments

- mesh** (`dolfin.Mesh`) The spatial domain (mesh)
- time** (`dolfin.Constant` or `None`) A constant holding the current time. If `None` is given, time is created for you, initialized to zero.
- model** (`cbcbeat.CardiacCellModel`) A representation of the cardiac cell model(s)

**I\_s (optional)** A typically time-dependent external stimulus given as a `dolphin.GenericFunction` or a `Markerwise`. NB: it is assumed that the time dependence of `I_s` is encoded via the 'time' Constant.

**params (dolphin.Parameters, optional)** Solver parameters

**static default\_parameters ()**

Initialize and return a set of default parameters

**Returns** A set of parameters (`dolphin.Parameters`)

**solution\_fields ()**

Return tuple of previous and current solution objects.

Modifying these will modify the solution objects of the solver and thus provides a way for setting initial conditions for instance.

**Returns** (previous vs, current vs) (`tuple of dolphin.Function`)

**solve (interval, dt=None)**

Solve the problem given by the model on a given time interval (t0, t1) with a given timestep dt and return generator for a tuple of the interval and the current vs solution.

**Arguments**

**interval (tuple)** The time interval for the solve given by (t0, t1)

**dt (int, optional)** The timestep for the solve. Defaults to length of interval

**Returns** (timestep, current vs) via (`genexpr`)

*Example of usage:*

```
# Create generator
solutions = solver.solve((0.0, 1.0), 0.1)

# Iterate over generator (computes solutions as you go)
for (interval, vs) in solutions:
    # do something with the solutions
```

**step (interval)**

Solve on the given time step (t0, t1).

End users are recommended to use `solve` instead.

**Arguments**

**interval (tuple)** The time interval (t0, t1) for the step

**time**

The internal time of the solver.

**class cbcbeat.cellsolver.CardiacODESolver (mesh, time, model, I\_s=None, params=None)**

Bases: `object`

An optimised solver for systems of ODEs typically encountered in cardiac applications of the form: find a scalar field  $v = v(x, t)$  and a vector field  $s = s(x, t)$

$$\begin{aligned}v_t &= -I_{ion}(v, s) + I_s \\s_t &= F(v, s)\end{aligned}$$

where  $I_{ion}$  and  $F$  are given non-linear functions, and  $I_s$  is some prescribed stimulus.

---

**Note:** For the sake of simplicity and consistency with other solver objects, this solver operates on its solution fields (as state variables) directly internally. More precisely, solve (and step) calls will act by updating the internal solution fields. It implies that initial conditions can be set (and are intended to be set) by modifying the solution fields prior to simulation.

---

### Arguments

**mesh** (`dolfin.Mesh`) The spatial mesh (mesh)

**time** (`dolfin.Constant` or `None`) A constant holding the current time. If `None` is given, time is created for you, initialized to zero.

**model** (`cbcbeat.CardiacCellModel`) A representation of the cardiac cell model(s)

**I<sub>s</sub>** (`dolfin.Expression`, optional) A typically time-dependent external stimulus. NB: it is assumed that the time dependence of I<sub>s</sub> is encoded via the ‘time’ Constant.

**params** (`dolfin.Parameters`, optional) Solver parameters

**static default\_parameters** ()

Initialize and return a set of default parameters

**Returns** A set of parameters (`dolfin.Parameters`)

**solution\_fields** ()

Return current solution object.

Modifying this will modify the solution object of the solver and thus provides a way for setting initial conditions for instance.

**Returns** (previous `vs_`, current `vs`) (`dolfin.Function`)

**solve** (*interval*, *dt=None*)

Solve the problem given by the model on a given time interval (t0, t1) with a given timestep dt and return generator for a tuple of the interval and the current vs solution.

### Arguments

**interval** (**tuple**) The time interval for the solve given by (t0, t1)

**dt** (**int**, optional) The timestep for the solve. Defaults to length of interval

**Returns** (timestep, current vs) via (`genexpr`)

*Example of usage:*

```
# Create generator
solutions = solver.solve((0.0, 1.0), 0.1)

# Iterate over generator (computes solutions as you go)
for (interval, vs) in solutions:
    # do something with the solutions
```

**step** (*interval*)

Solve on the given time step (t0, t1).

End users are recommended to use solve instead.

### Arguments

**interval** (**tuple**) The time interval (t0, t1) for the step

```
class cbcbeat.cellsolver.SingleCellSolver (model, time, params=None)  
    Bases: cbcbeat.cellsolver.CardiacODESolver
```

### cbcbeat.dolfinimport module

This module handles all dolfin import in cbcbeat. Here dolfin and dolfin\_adjoint gets imported. If dolfin\_adjoint is not present it will not be imported.

### cbcbeat.gossplittingsolver module

This module contains a CellSolver that uses JIT compiled Gotran models together with GOSS (General ODE System Solver), which can be interfaced by the GossSplittingSolver

```
class cbcbeat.gossplittingsolver.GOSSplittingSolver (model, params=None)
```

```
static default_parameters ()
```

Initialize and return a set of default parameters for the splitting solver

**Returns** A set of parameters (*dolfin.Parameters*)

To inspect all the default parameters, do:

```
info(SplittingSolver.default_parameters(), True)
```

```
merge (solution)
```

Extract membrane potential from solutions from the PDE solve and put it into membrane potential used for the ODE solve.

**Arguments**

**solution** (*dolfin.Function*) Function holding the combined result

```
solution_fields ()
```

Return tuple of previous and current solution objects.

Modifying these will modify the solution objects of the solver and thus provides a way for setting initial conditions for instance.

**Returns** (current v, current vur) (*tuple of dolfin.Function*)

```
solve (interval, dt)
```

Solve the problem given by the model on a given time interval (t0, t1) with a given timestep dt and return generator for a tuple of the time step and the solution fields.

**Arguments**

**interval** (*tuple*) The time interval for the solve given by (t0, t1)

**dt** (*int*) The timestep for the solve

**Returns** (timestep, solution\_fields) via (*genexpr*)

*Example of usage:*

```
# Create generator  
solutions = solver.solve((0.0, 1.0), 0.1)  
  
# Iterate over generator (computes solutions as you go)  
for ((t0, t1), (v, vur)) in solutions:  
    # do something with the solutions
```



**step** (*interval*)

Solve the problem given by the model on a given time interval (t0, t1) with timestep given by the interval length.

**Arguments**

**interval (tuple)** The time interval for the solve given by (t0, t1)

**Invariants** Given self.\_vs in a correct state at t0, provide v and s (in self.vs) and u (in self.vur) in a correct state at t1. (Note that self.vur[0] == self.vs[0] only if theta = 1.0.)

## cbcbeat.gotran2cellmodel module

## cbcbeat.gotran2dolphin module

**class** cbcbeat.gotran2dolphin.**DOLFINCodeGenerator** (*code\_params=None*)

Bases: PythonCodeGenerator

Class for generating a DOLFIN compatible declarations of an ODE from a gotran file

**static default\_parameters** ()

**function\_code** (*comp, indent=0, default\_arguments=None, include\_signature=True*)

**init\_parameters\_code** (*ode, indent=0*)

Generate code for setting parameters

**init\_states\_code** (*ode, indent=0*)

Generate code for setting initial condition

## cbcbeat.markerwisefield module

**class** cbcbeat.markerwisefield.**Markerwise** (*objects, keys, markers*)

Bases: object

A container class representing an object defined by a number of objects combined with a mesh function defining mesh domains and a map between the these.

**Arguments**

**objects (tuple)** the different objects

**keys (tuple of ints)** a map from the objects to the domains marked in markers

**markers (dolfin.MeshFunction)** a mesh function mapping which domains the mesh consist of

*Example of usage*

Given (g0, g1), (2, 5) and markers, let

g = g0 on domains marked by 2 in markers g = g1 on domains marked by 5 in markers

letting:

```
g = Markerwise((g0, g1), (2, 5), markers)
```

**keys** ()

The keys or domain numbers

**markers** ()

The markers

**values ()**  
The objects

`cbcbeat.markerwisefield.handle_markerwise (g, classtype)`

`cbcbeat.markerwisefield.rhs_with_markerwise_field (g, mesh, v)`

### cbcbeat.monodomainsolver module

These solvers solve the (pure) monodomain equations on the form: find the transmembrane potential  $v = v(x, t)$  such that

$$v_t - \text{div}(G_i v) = I_s$$

where the subscript  $t$  denotes the time derivative;  $G_i$  denotes a weighted gradient:  $G_i = M_i \text{grad}(v)$  for, where  $M_i$  is the intracellular cardiac conductivity tensor;  $I_s$  is prescribed input. In addition, initial conditions are given for  $v$ :

$$v(x, 0) = v_0$$

Finally, boundary conditions must be prescribed. For now, this solver assumes pure homogeneous Neumann boundary conditions for  $v$ .

**class** `cbcbeat.monodomainsolver.BasicMonodomainSolver (mesh, time, M_i, I_s=None, v_=None, params=None)`

Bases: `object`

This solver is based on a theta-scheme discretization in time and CG\_1 elements in space.

---

**Note:** For the sake of simplicity and consistency with other solver objects, this solver operates on its solution fields (as state variables) directly internally. More precisely, `solve` (and `step`) calls will act by updating the internal solution fields. It implies that initial conditions can be set (and are intended to be set) by modifying the solution fields prior to simulation.

---

#### Arguments

**mesh (dolfin.Mesh)** The spatial domain (mesh)

**time (dolfin.Constant or None)** A constant holding the current time. If `None` is given, time is created for you, initialized to zero.

**M\_i (ufl.Expr)** The intracellular conductivity tensor (as an UFL expression)

**I\_s (dict, optional)** A typically time-dependent external stimulus given as a dict, with domain markers as the key and a `dolfin.Expression` as values. NB: it is assumed that the time dependence of `I_s` is encoded via the 'time' Constant.

**v\_ (ufl.Expr, optional)** Initial condition for  $v$ . A new `dolfin.Function` will be created if none is given.

**params (dolfin.Parameters, optional)** Solver parameters

**static default\_parameters ()**  
Initialize and return a set of default parameters

**Returns** A set of parameters (`dolfin.Parameters`)

To inspect all the default parameters, do:

```
info(BasicMonodomainSolver.default_parameters(), True)
```

**solution\_fields()**

Return tuple of previous and current solution objects.

Modifying these will modify the solution objects of the solver and thus provides a way for setting initial conditions for instance.

**Returns** (previous v, current v) (tuple of `dolfin.Function`)

**solve(interval, dt=None)**

Solve the discretization on a given time interval (t0, t1) with a given timestep dt and return generator for a tuple of the interval and the current solution.

**Arguments**

**interval (tuple)** The time interval for the solve given by (t0, t1)

**dt (int, optional)** The timestep for the solve. Defaults to length of interval

**Returns** (timestep, solution\_field) via (genexpr)

*Example of usage:*

```
# Create generator
solutions = solver.solve((0.0, 1.0), 0.1)

# Iterate over generator (computes solutions as you go)
for (interval, solution_fields) in solutions:
    (t0, t1) = interval
    v_, v = solution_fields
    # do something with the solutions
```

**step(interval)**

Solve on the given time interval (t0, t1).

**Arguments**

**interval (tuple)** The time interval (t0, t1) for the step

**Invariants** Assuming that `v_` is in the correct state for t0, gives `self.v` in correct state at t1.

**time**

The internal time of the solver.

**class** `cbcbeat.monodomainsolver.MonodomainSolver` (*mesh, time, M\_i, I\_s=None, v\_=None, params=None*)

Bases: `cbcbeat.monodomainsolver.BasicMonodomainSolver`

This solver is based on a theta-scheme discretization in time and CG\_1 elements in space.

---

**Note:** For the sake of simplicity and consistency with other solver objects, this solver operates on its solution fields (as state variables) directly internally. More precisely, `solve` (and `step`) calls will act by updating the internal solution fields. It implies that initial conditions can be set (and are intended to be set) by modifying the solution fields prior to simulation.

---

**Arguments**

**mesh (dolfin.Mesh)** The spatial domain (mesh)

**time** (`dolfin.Constant` or `None`) A constant holding the current time. If `None` is given, time is created for you, initialized to zero.

**M<sub>i</sub>** (`ufl.Expr`) The intracellular conductivity tensor (as an UFL expression)

**I<sub>s</sub>** (`dict`, `optional`) A typically time-dependent external stimulus given as a dict, with domain markers as the key and a `dolfin.Expression` as values. NB: it is assumed that the time dependence of `Is` is encoded via the ‘time’ Constant.

**v<sub>0</sub>** (`ufl.Expr`, `optional`) Initial condition for `v`. A new `dolfin.Function` will be created if none is given.

**params** (`dolfin.Parameters`, `optional`) Solver parameters

**static default\_parameters** ()

Initialize and return a set of default parameters

**Returns** A set of parameters (`dolfin.Parameters`)

To inspect all the default parameters, do:

```
info(MonodomainSolver.default_parameters(), True)
```

**linear\_solver**

The linear solver (`dolfin.LUSolver` or `dolfin.KrylovSolver`).

**step** (`interval`)

Solve on the given time step (`t0`, `t1`).

**Arguments**

**interval** (`tuple`) The time interval (`t0`, `t1`) for the step

**Invariants** Assuming that `v0` is in the correct state for `t0`, gives `self.v` in correct state at `t1`.

**variational\_forms** (`kn`)

Create the variational forms corresponding to the given discretization of the given system of equations.

**Arguments**

**k<sub>n</sub>** (`ufl.Expr` or `float`) The time step

**Returns** (`lhs`, `rhs`, `prec`) (`tuple` of `ufl.Form`)

## cbcbeat.splittingsolver module

This module contains splitting solvers for `CardiacModel` objects. In particular, the classes

- `SplittingSolver`
- `BasicSplittingSolver`

These solvers solve the bidomain (or monodomain) equations on the form: find the transmembrane potential  $v = v(x, t)$  in mV, the extracellular potential  $u = u(x, t)$  in mV, and any additional state variables  $s = s(x, t)$  such that

$$\begin{aligned} v_t - \operatorname{div}(M_i \operatorname{grad} v + M_i \operatorname{grad} u) &= -I_{ion}(v, s) + I_s \\ \operatorname{div}(M_i \operatorname{grad} v + (M_i + M_e) \operatorname{grad} u) &= I_a \\ s_t &= F(v, s) \end{aligned}$$

where

- the subscript  $t$  denotes the time derivative,

- $M_i$  and  $M_e$  are conductivity tensors (in  $\text{mm}^2/\text{ms}$ )
- $I_s$  is prescribed input current (in  $\text{mV}/\text{ms}$ )
- $I_a$  is prescribed input current (in  $\text{mV}/\text{ms}$ )
- $I_{ion}$  and  $F$  are typically specified by a cell model

**Note that  $M_i$  and  $M_e$  can be viewed as scaled by  $\chi * C_m$  where**

- $\chi$  is the surface-to volume ratio of cells (in  $1/\text{mm}$ ),
- $C_m$  is the specific membrane capacitance (in  $\mu\text{F}/(\text{mm}^2)$ ),

In addition, initial conditions are given for  $v$  and  $s$ :

$$\begin{aligned}v(x, 0) &= v_0 \\s(x, 0) &= s_0\end{aligned}$$

Finally, boundary conditions must be prescribed. These solvers assume pure Neumann boundary conditions for  $v$  and  $u$  and enforce the additional average value zero constraint for  $u$ .

The solvers take as input a `CardiacModel` providing the required input specification of the problem. In particular, the applied current  $I_a$  is extracted from the `applied_current` attribute, while the stimulus  $I_s$  is extracted from the `stimulus` attribute.

It should be possible to use the solvers interchangeably. However, note that the `BasicSplittingSolver` is not optimised and should be used for testing or debugging purposes primarily.

**class** `cbcbeat.splittingsolver.SplittingSolver` (*model*, *params=None*)

Bases: `cbcbeat.splittingsolver.BasicSplittingSolver`

An optimised solver for the bidomain equations based on the operator splitting scheme described in Sundnes et al 2006, p. 78 ff.

The solver computes as solutions:

- “vs” (`dolfin.Function`) representing the solution for the transmembrane potential and any additional state variables, and
- “vur” (`dolfin.Function`) representing the transmembrane potential in combination with the extracellular potential and an additional Lagrange multiplier.

The algorithm can be controlled by a number of parameters. In particular, the splitting algorithm can be controlled by the parameter “theta”: “theta” set to 1.0 corresponds to a (1st order) Godunov splitting while “theta” set to 0.5 to a (2nd order) Strang splitting.

#### Arguments

**model** (`cbcbeat.cardiacmodels.CardiacModel`) a `CardiacModel` object describing the simulation set-up

**params** (`dolfin.Parameters`, optional) a `Parameters` object controlling solver parameters

*Example of usage:*

```
from cbcbeat import *

# Describe the cardiac model
mesh = UnitSquareMesh(100, 100)
time = Constant(0.0)
cell_model = FitzHughNagumoManual()
stimulus = Expression("10*t*x[0]", t=time, degree=1)
cm = CardiacModel(mesh, time, 1.0, 1.0, cell_model, stimulus)
```

(continues on next page)

(continued from previous page)

```

# Extract default solver parameters
ps = SplittingSolver.default_parameters()

# Examine the default parameters
info(ps, True)

# Update parameter dictionary
ps["theta"] = 1.0 # Use first order splitting
ps["CardiacODESolver"]["scheme"] = "GRL1" # Use Generalized Rush-Larsen scheme

ps["pde_solver"] = "monodomain" # Use monodomain_
↪equations as the PDE model
ps["MonodomainSolver"]["linear_solver_type"] = "direct" # Use direct linear_
↪solver of the PDEs
ps["MonodomainSolver"]["theta"] = 1.0 # Use backward Euler for_
↪temporal discretization for the PDEs

solver = SplittingSolver(cm, params=ps)

# Extract the solution fields and set the initial conditions
(vs_, vs, vur) = solver.solution_fields()
vs_.assign(cell_model.initial_conditions())

# Solve
dt = 0.1
T = 1.0
interval = (0.0, T)
for (timestep, fields) in solver.solve(interval, dt):
    (vs_, vs, vur) = fields
    # Do something with the solutions

```

### Assumptions

- The cardiac conductivities do not vary in time

### static default\_parameters()

Initialize and return a set of default parameters for the splitting solver

**Returns** The set of default parameters (`dofin.Parameters`)

*Example of usage:*

```
info(SplittingSolver.default_parameters(), True)
```

### class cbcbeat.splittingsolver.BasicSplittingSolver(model, params=None)

A non-optimised solver for the bidomain equations based on the operator splitting scheme described in Sundnes et al 2006, p. 78 ff.

The solver computes as solutions:

- “vs” (`dofin.Function`) representing the solution for the transmembrane potential and any additional state variables, and
- “vur” (`dofin.Function`) representing the transmembrane potential in combination with the extracellular potential and an additional Lagrange multiplier.

The algorithm can be controlled by a number of parameters. In particular, the splitting algorithm can be controlled by the parameter “theta”: “theta” set to 1.0 corresponds to a (1st order) Godunov splitting while “theta”

set to 0.5 to a (2nd order) Strang splitting.

This solver has not been optimised for computational efficiency and should therefore primarily be used for debugging purposes. For an equivalent, but more efficient, solver, see `cbcbeat.splittingsolver.SplittingSolver`.

#### Arguments

- model** (`cbcbeat.cardiacmodels.CardiacModel`) a CardiacModel object describing the simulation set-up
- params** (`dolfin.Parameters`, optional) a Parameters object controlling solver parameters

#### Assumptions

- The cardiac conductivities do not vary in time

#### static default\_parameters ()

Initialize and return a set of default parameters for the splitting solver

**Returns** A set of parameters (`dolfin.Parameters`)

To inspect all the default parameters, do:

```
info(BasicSplittingSolver.default_parameters(), True)
```

#### merge (solution)

Combine solutions from the PDE solve and the ODE solve to form a single mixed function.

#### Arguments

- solution** (`dolfin.Function`) Function holding the combined result

#### solution\_fields ()

Return tuple of previous and current solution objects.

Modifying these will modify the solution objects of the solver and thus provides a way for setting initial conditions for instance.

**Returns** (previous vs, current vs, current vur) (tuple of `dolfin.Function`)

#### solve (interval, dt)

Solve the problem given by the model on a given time interval (t0, t1) with a given timestep dt and return generator for a tuple of the time step and the solution fields.

#### Arguments

- interval** (**tuple**) The time interval for the solve given by (t0, t1)
- dt** (**int, list of tuples of floats**) The timestep for the solve. A list of tuples of floats can also be passed. Each tuple should contain two floats where the first includes the start time and the second the dt.

**Returns** (timestep, solution\_fields) via (genexpr)

*Example of usage:*

```
# Create generator
dts = [(0., 0.1), (1.0, 0.05), (2.0, 0.1)]
solutions = solver.solve((0.0, 1.0), dts)

# Iterate over generator (computes solutions as you go)
for ((t0, t1), (vs_, vs, vur)) in solutions:
    # do something with the solutions
```

**step** (*interval*)

Solve the problem given by the model on a given time interval (t0, t1) with timestep given by the interval length.

**Arguments**

**interval (tuple)** The time interval for the solve given by (t0, t1)

**Invariants** Given self.\_vs in a correct state at t0, provide v and s (in self.vs) and u (in self.vur) in a correct state at t1. (Note that self.vur[0] == self.vs[0] only if theta = 1.0.)

## cbcbeat.utils module

This module provides various utilities for internal use.

`cbcbeat.utils.state_space` (*domain, d, family=None, k=1*)

Return function space for the state variables.

**Arguments**

**domain (dolfin.Mesh)** The computational domain

**d (int)** The number of states

**family (string, optional)** The finite element family, defaults to “CG” if None is given.

**k (int, optional)** The finite element degree, defaults to 1

**Returns** a function space (`dolfin.FunctionSpace`)

`cbcbeat.utils.end_of_time` (*T, t0, t1, dt*)

Return True if the interval (t0, t1) is the last before the end time T, otherwise False.

`cbcbeat.utils.convergence_rate` (*hs, errors*)

Compute and return rates of convergence  $r_i$  such that

$$errors = Chs^r$$

**class** `cbcbeat.utils.Projecter` (*V, params=None*)

Bases: `object`

Customized class for repeated projection.

**Arguments**

**V (dolfin.FunctionSpace)** The function space to project into

**solver\_type (string, optional)** “iterative” (default) or “direct”

**Example of usage::** `my_project = Projecter(V, solver_type=“direct”) u = Function(V) f = Function(W)`  
`my_project(f, u)`

**static default\_parameters** ()

## Module contents

The cbcbeat Python module is a problem and solver collection for cardiac electrophysiology models.

To import the module, type:

```
from cbcbeat import *
```



# CHAPTER 7

---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`



## C

cbcbeat, 36  
cbcbeat.bidomainsolver, 21  
cbcbeat.cardiacmodels, 23  
cbcbeat.cellmodels, 21  
cbcbeat.cellmodels.beeler\_reuter\_1977,  
13  
cbcbeat.cellmodels.cardiaccellmodel, 14  
cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered,  
15  
cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered,  
16  
cbcbeat.cellmodels.fitzhughnagumo, 16  
cbcbeat.cellmodels.fitzhughnagumo\_manual,  
16  
cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010,  
17  
cbcbeat.cellmodels.nocellmodel, 17  
cbcbeat.cellmodels.rogers\_mcculloch\_manual,  
18  
cbcbeat.cellmodels.tentusscher\_2004\_mcell,  
18  
cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont,  
19  
cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc,  
19  
cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell,  
20  
cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell,  
20  
cbcbeat.cellsolver, 24  
cbcbeat.dolphinimport, 28  
cbcbeat.gossplittingsolver, 28  
cbcbeat.gotran2cellmodel, 29  
cbcbeat.gotran2dolphin, 29  
cbcbeat.markerwisefield, 29  
cbcbeat.monodomainsolver, 30  
cbcbeat.splittingsolver, 32  
cbcbeat.utils, 36



**A**

applied\_current() (cbcbeat.cardiacmodels.CardiacModel method), 24

**B**

BasicBidomainSolver (class in cbcbeat.bidomainsolver), 21

BasicCardiacODESolver (class in cbcbeat.cellsolver), 25

BasicMonodomainSolver (class in cbcbeat.monodomainsolver), 30

BasicSingleCellSolver (class in cbcbeat.cellsolver), 24

BasicSplittingSolver (class in cbcbeat.splittingsolver), 34

Beeler\_reuter\_1977 (class in cbcbeat.cellmodels.beeler\_reuter\_1977), 13

BidomainSolver (class in cbcbeat.bidomainsolver), 22

**C**

CardiacCellModel (class in cbcbeat.cellmodels.cardiaccellmodel), 14

CardiacModel (class in cbcbeat.cardiacmodels), 23

CardiacODESolver (class in cbcbeat.cellsolver), 26

cbcbeat (module), 36

cbcbeat.bidomainsolver (module), 21

cbcbeat.cardiacmodels (module), 23

cbcbeat.cellmodels (module), 21

cbcbeat.cellmodels.beeler\_reuter\_1977 (module), 13

cbcbeat.cellmodels.cardiaccellmodel (module), 14

cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered (module), 15

cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered (module), 16

cbcbeat.cellmodels.fitzhughnagumo (module), 16

cbcbeat.cellmodels.fitzhughnagumo\_manual (module), 16

cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010 (module), 17

cbcbeat.cellmodels.nocellmodel (module), 17

cbcbeat.cellmodels.rogers\_mcculloch\_manual (module), 18

cbcbeat.cellmodels.tentusscher\_2004\_mcell (module), 18

cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont (module), 19

cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc (module), 19

cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell (module), 20

cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell (module), 20

cbcbeat.cellsolver (module), 24

cbcbeat.dolfinimport (module), 28

cbcbeat.gossplittingsolver (module), 28

cbcbeat.gotran2cellmodel (module), 29

cbcbeat.gotran2dolphin (module), 29

cbcbeat.markerwisefield (module), 29

cbcbeat.monodomainsolver (module), 30

cbcbeat.splittingsolver (module), 32

cbcbeat.utils (module), 36

cell\_models() (cbcbeat.cardiacmodels.CardiacModel method), 24

conductivities() (cbcbeat.cardiacmodels.CardiacModel method), 24

convergence\_rate() (in module cbcbeat.utils), 36

**D**

default\_initial\_conditions() (cbcbeat.cellmodels.beeler\_reuter\_1977.Beeler\_reuter\_1977 static method), 14

default\_initial\_conditions() (cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel static method), 14

default\_initial\_conditions() (cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered.Fenton\_karma\_1998\_BR\_altered static method), 15

default\_initial\_conditions() (cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered.Fenton\_karma\_1998\_MLR1\_altered static method), 16

default\_initial\_conditions() (cbcbeat.cellmodels.fitzhughnagumo.Fitzhughnagumo static method), 16  
 default\_initial\_conditions() (cbcbeat.cellmodels.fitzhughnagumo\_manual.FitzHughNagumoManual static method), 17  
 default\_initial\_conditions() (cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010.Grandi\_pasqualini\_bers\_2010 static method), 17  
 default\_initial\_conditions() (cbcbeat.cellmodels.nocellmodel.NoCellModel static method), 17  
 default\_initial\_conditions() (cbcbeat.cellmodels.rogers\_mcculloch\_manual.RogersMcCullochManual static method), 18  
 default\_initial\_conditions() (cbcbeat.cellmodels.tentusscher\_2004\_mcell.Tentusscher\_2004\_mcell static method), 19  
 default\_initial\_conditions() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont.Tentusscher\_2004\_mcell\_cont static method), 19  
 default\_initial\_conditions() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc.Defaultspher\_2004\_mcell\_disc static method), 19  
 default\_initial\_conditions() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell.Tentusscher\_panfilov\_2006\_epi\_cell static method), 20  
 default\_initial\_conditions() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell.Tentusscher\_panfilov\_2006\_M\_cell static method), 20  
 default\_parameters() (cbcbeat.bidomainsolver.BasicBidomainSolver static method), 21  
 default\_parameters() (cbcbeat.bidomainsolver.BidomainSolver static method), 23  
 default\_parameters() (cbcbeat.cellmodels.beeler\_reuter\_1977.Beeler\_reuter\_1977 static method), 14  
 default\_parameters() (cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel static method), 14  
 default\_parameters() (cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered.Fenton\_karma\_1998\_BR\_altered static method), 15  
 default\_parameters() (cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered.Fenton\_karma\_1998\_MLR1\_altered static method), 16  
 default\_parameters() (cbcbeat.cellmodels.fitzhughnagumo.Fitzhughnagumo static method), 16  
 default\_parameters() (cbcbeat.cellmodels.fitzhughnagumo\_manual.FitzHughNagumoManual static method), 17  
 default\_parameters() (cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010.Grandi\_pasqualini\_bers\_2010 static method), 17  
 default\_parameters() (cbcbeat.cellmodels.rogers\_mcculloch\_manual.RogersMcCullochManual static method), 18  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_2004\_mcell.Tentusscher\_2004\_mcell static method), 19  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont.Tentusscher\_2004\_mcell\_cont static method), 19  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc.Tentusscher\_2004\_mcell\_disc static method), 19  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell.Tentusscher\_panfilov\_2006\_epi\_cell static method), 20  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell.Tentusscher\_panfilov\_2006\_M\_cell static method), 20  
 default\_parameters() (cbcbeat.bidomainsolver.BasicBidomainSolver static method), 21  
 default\_parameters() (cbcbeat.bidomainsolver.BidomainSolver static method), 23  
 default\_parameters() (cbcbeat.cellmodels.beeler\_reuter\_1977.Beeler\_reuter\_1977 static method), 14  
 default\_parameters() (cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel static method), 14  
 default\_parameters() (cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered.Fenton\_karma\_1998\_BR\_altered static method), 15  
 default\_parameters() (cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered.Fenton\_karma\_1998\_MLR1\_altered static method), 16  
 default\_parameters() (cbcbeat.cellmodels.fitzhughnagumo.Fitzhughnagumo static method), 16  
 default\_parameters() (cbcbeat.cellmodels.fitzhughnagumo\_manual.FitzHughNagumoManual static method), 17  
 default\_parameters() (cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010.Grandi\_pasqualini\_bers\_2010 static method), 17  
 default\_parameters() (cbcbeat.cellmodels.rogers\_mcculloch\_manual.RogersMcCullochManual static method), 18  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_2004\_mcell.Tentusscher\_2004\_mcell static method), 19  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont.Tentusscher\_2004\_mcell\_cont static method), 19  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc.Tentusscher\_2004\_mcell\_disc static method), 19  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell.Tentusscher\_panfilov\_2006\_epi\_cell static method), 20  
 default\_parameters() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell.Tentusscher\_panfilov\_2006\_M\_cell static method), 20  
 default\_parameters() (cbcbeat.monodomainsolver.BasicMonodomainSolver static method), 30  
 default\_parameters() (cbcbeat.monodomainsolver.MonodomainSolver static method), 32  
 default\_parameters() (cbcbeat.splittingsolver.BasicSplittingSolver static method), 34  
 default\_parameters() (cbcbeat.splittingsolver.SplittingSolver static method), 34  
 default\_parameters() (cbcbeat.gossplittingsolver.GOSSplittingSolver static method), 28  
 default\_parameters() (cbcbeat.gotran2dolphin.DOLFINCodeGenerator static method), 29  
 default\_parameters() (cbcbeat.gotran2dolphin.DOLFINCodeGenerator (class in cbcbeat.gotran2dolphin), 29  
 domain() (cbcbeat.cardiacmodels.CardiacModel method), 24  
 extracellular\_conductivity() (cbcbeat.cardiacmodels.CardiacModel method), 24  
 find\_opt\_time() (in module cbcbeat.utils), 36  
 F  
 f() (cbcbeat.cellmodels.beeler\_reuter\_1977.Beeler\_reuter\_1977 method), 13  
 f() (cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered.Fenton\_karma\_1998\_BR\_altered method), 14  
 f() (cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered.Fenton\_karma\_1998\_MLR1\_altered method), 15  
 f() (cbcbeat.cellmodels.fitzhughnagumo.Fitzhughnagumo method), 15  
 f() (cbcbeat.cellmodels.fitzhughnagumo\_manual.FitzHughNagumoManual method), 16  
 f() (cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered.Fenton\_karma\_1998\_BR\_altered method), 15  
 f() (cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered.Fenton\_karma\_1998\_MLR1\_altered method), 16  
 f() (cbcbeat.cellmodels.fitzhughnagumo.Fitzhughnagumo method), 16  
 f() (cbcbeat.cellmodels.fitzhughnagumo\_manual.FitzHughNagumoManual method), 17  
 f() (cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010.Grandi\_pasqualini\_bers\_2010 method), 17  
 f() (cbcbeat.cellmodels.rogers\_mcculloch\_manual.RogersMcCullochManual method), 17  
 f() (cbcbeat.cellmodels.tentusscher\_2004\_mcell.Tentusscher\_2004\_mcell method), 17  
 f() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont.Tentusscher\_2004\_mcell\_cont method), 17  
 f() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc.Tentusscher\_2004\_mcell\_disc method), 17  
 f() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell.Tentusscher\_panfilov\_2006\_epi\_cell method), 20  
 f() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell.Tentusscher\_panfilov\_2006\_M\_cell method), 20  
 f() (cbcbeat.cellmodels.nocellmodel.NoCellModel method), 17

F) (cbcbeat.cellmodels.rogers\_mcculloch\_manual.RogersMcCulloch (cbcbeat.cellmodels.nocellmodel.NoCellModel method), 18  
 method), 17  
 F) (cbcbeat.cellmodels.tentusscher\_2004\_mcell.Tentusscher2004\_mcell.cbcbeat.cellmodels.rogers\_mcculloch\_manual.RogersMcCulloch method), 18  
 method), 18  
 F) (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont.Tentusscher2004\_mcell\_cont.cbcbeat.cellmodels.tentusscher\_2004\_mcell.Tentusscher\_2004\_mcell method), 19  
 method), 19  
 F) (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc.Tentusscher2004\_mcell\_disc.cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont.Tentusscher\_2004\_m method), 19  
 method), 19  
 F) (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell.TentusscherPanfilov2006\_epi\_cell.cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc.Tentusscher\_2004\_m method), 20  
 method), 19  
 F) (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell.TentusscherPanfilov2006\_M\_cell.cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell.Tentusscher\_pa method), 20  
 method), 20  
 Fenton\_karma\_1998\_BR\_altered (class in I) (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell.Tentusscher\_pan method), 20  
 cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered), method), 20  
 15 init\_parameters\_code() (cbcbeat.gotran2dofin.DOLFINCodeGenerator  
 method), 29  
 Fenton\_karma\_1998\_MLR1\_altered (class in method), 29  
 cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered), method), 29  
 16 intricates\_code() (cbcbeat.gotran2dofin.DOLFINCodeGenerator  
 method), 29  
 Fitzhughnagumo (class in initial\_conditions() (cbcbeat.cellmodels.cardiaccellmodel.Cardiaccellmodel.Cardiaccellmodel method), 16  
 cbcbeat.cellmodels.fitzhughnagumo), 16 method), 14  
 FitzHughNagumoManual (class in initial\_conditions() (cbcbeat.cellmodels.cardiaccellmodel.MultiCellModel method), 15  
 cbcbeat.cellmodels.fitzhughnagumo\_manual), method), 15  
 16 intracellular\_conductivity()  
 function\_code() (cbcbeat.gotran2dofin.DOLFINCodeGenerator (cbcbeat.cardiacmodels.Cardiaccellmodel.Cardiaccellmodel method), 29  
 method), 29 method), 24

## G

GOSSplittingSolver (class in cbcbeat.gossplittingsolver),  
 28  
 Grandi\_pasqualini\_bers\_2010 (class in  
 cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010),  
 17

## H

handle\_markerwise() (in  
 cbcbeat.markerwisefield), 30

## I

I) (cbcbeat.cellmodels.beeler\_reuter\_1977.Beeler\_reuter\_1977 method), 14  
 I) (cbcbeat.cellmodels.cardiaccellmodel.Cardiaccellmodel method), 14  
 I) (cbcbeat.cellmodels.cardiaccellmodel.MultiCellModel method), 15  
 I) (cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered.Fenton\_karma\_1998\_BR\_altered method), 15  
 I) (cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered.Fenton\_karma\_1998\_MLR1\_altered method), 16  
 I) (cbcbeat.cellmodels.fitzhughnagumo.Fitzhughnagumo method), 16  
 I) (cbcbeat.cellmodels.fitzhughnagumo\_manual.FitzHughNagumoManual method), 17  
 I) (cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010.Grandi\_pasqualini\_bers\_2010 method), 17

## K

keys() (cbcbeat.cellmodels.cardiaccellmodel.MultiCellModel  
 method), 15

keys() (cbcbeat.markerwisefield.Markerwise method), 29

## L

linear\_solver (cbcbeat.bidomainsolver.BidomainSolver  
 attribute), 23

linear\_solver (cbcbeat.monodomainsolver.MonodomainSolver  
 attribute), 32

## M

markers() (cbcbeat.cellmodels.cardiaccellmodel.MultiCellModel  
 method), 15

markers() (cbcbeat.markerwisefield.Markerwise method),  
 29

Markerwise (class in cbcbeat.markerwisefield), 29

merge() (cbcbeat.gossplittingsolver.GOSSplittingSolver  
 method), 29

merge() (cbcbeat.splittingsolver.BasicSplittingSolver  
 method), 31

mesh() (cbcbeat.cellmodels.cardiaccellmodel.MultiCellModel  
 method), 15

models() (cbcbeat.cellmodels.cardiaccellmodel.MultiCellModel  
 method), 15

MonodomainSolver (class in  
 cbcbeat.bidomainsolver), 31

MultiCellModel (class in cbcbeat.cellmodels.cardiaccellmodel), 15

**N**

NoCellModel (class in cbcbeat.cellmodels.nocellmodel), 17

nullspace (cbcbeat.bidomainsolver.BidomainSolver attribute), 23

num\_models() (cbcbeat.cellmodels.cardiaccellmodel.MultiCellModel method), 15

num\_states() (cbcbeat.cellmodels.beeler\_reuter\_1977.Beeler\_reuter\_1977 method), 14

num\_states() (cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel method), 15

num\_states() (cbcbeat.cellmodels.cardiaccellmodel.MultiCellModel method), 15

num\_states() (cbcbeat.cellmodels.fenton\_karma\_1998\_BR\_altered.Fenton\_karma\_1998\_BR\_altered method), 15

num\_states() (cbcbeat.cellmodels.fenton\_karma\_1998\_MLR1\_altered.Fenton\_karma\_1998\_MLR1\_altered method), 16

num\_states() (cbcbeat.cellmodels.fitzhughnagumo.Fitzhughnagumo method), 16

num\_states() (cbcbeat.cellmodels.fitzhughnagumo\_manual.FitzHughNagumoManual method), 17

num\_states() (cbcbeat.cellmodels.grandi\_pasqualini\_bers\_2010.GrandiPasqualiniBers2010 method), 17

num\_states() (cbcbeat.cellmodels.nocellmodel.NoCellModel method), 18

num\_states() (cbcbeat.cellmodels.rogers\_mcculloch\_manual.RogersMcCullochManual method), 18

num\_states() (cbcbeat.cellmodels.tentusscher\_2004\_mcell.Tentusscher2004MCell method), 19

num\_states() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_comp.Tentusscher2004MCellComp method), 19

num\_states() (cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disp.Tentusscher2004MCellDisp method), 20

num\_states() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epicard.TentusscherPanfilov2006Epicard method), 20

num\_states() (cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M1cell.TentusscherPanfilov2006M1Cell method), 20

**P**

parameters() (cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel method), 15

Projecter (class in cbcbeat.utils), 36

**R**

rhs\_with\_markerwise\_field() (in module cbcbeat.markerwisefield), 30

RogersMcCulloch (class in cbcbeat.cellmodels.rogers\_mcculloch\_manual), 18

**S**

set\_initial\_conditions() (cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel method), 15

set\_parameters() (cbcbeat.cellmodels.cardiaccellmodel.CardiacCellModel method), 15

SingleCellSolver (class in cbcbeat.cellsolver), 27

solution\_fields() (cbcbeat.bidomainsolver.BasicBidomainSolver method), 22

solution\_fields() (cbcbeat.cellsolver.BasicCardiacODESolver method), 26

solution\_fields() (cbcbeat.cellsolver.CardiacODESolver method), 27

solution\_fields() (cbcbeat.gossplittingsolver.GOSSplittingSolver method), 28

solution\_fields() (cbcbeat.monodomainsolver.BasicMonodomainSolver method), 31

solution\_fields() (cbcbeat.splittingsolver.BasicSplittingSolver method), 35

solve() (cbcbeat.bidomainsolver.BasicBidomainSolver method), 22

solve() (cbcbeat.cellsolver.BasicCardiacODESolver method), 26

solve() (cbcbeat.cellsolver.CardiacODESolver method), 27

solve() (cbcbeat.gossplittingsolver.GOSSplittingSolver method), 28

solve() (cbcbeat.monodomainsolver.BasicMonodomainSolver method), 31

solve() (cbcbeat.splittingsolver.BasicSplittingSolver method), 35

splitting\_solver() (class in cbcbeat.splittingsolver), 33

state\_space() (in module cbcbeat.utils), 36

step() (cbcbeat.bidomainsolver.BasicBidomainSolver method), 22

step() (cbcbeat.monodomainsolver.BasicMonodomainSolver method), 23

step() (cbcbeat.cellsolver.BasicCardiacODESolver method), 26

step() (cbcbeat.gossplittingsolver.GOSSplittingSolver method), 28

step() (cbcbeat.monodomainsolver.MonodomainSolver method), 32

step() (cbcbeat.splittingsolver.BasicSplittingSolver method), 35

stimulus() (cbcbeat.cardiacmodels.CardiacModel method), 24

**T**

Tentusscher\_2004\_mcell (class in cbcbeat.cellmodels.tentusscher\_2004\_mcell), 18



Tentusscher\_2004\_mcell\_cont (class in  
cbcbeat.cellmodels.tentusscher\_2004\_mcell\_cont),  
19

Tentusscher\_2004\_mcell\_disc (class in  
cbcbeat.cellmodels.tentusscher\_2004\_mcell\_disc),  
19

Tentusscher\_panfilov\_2006\_epi\_cell (class in  
cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_epi\_cell),  
20

Tentusscher\_panfilov\_2006\_M\_cell (class in  
cbcbeat.cellmodels.tentusscher\_panfilov\_2006\_M\_cell),  
20

time (cbcbeat.bidomainsolver.BasicBidomainSolver at-  
tribute), 22

time (cbcbeat.cellsolver.BasicCardiacODESolver at-  
tribute), 26

time (cbcbeat.monodomainsolver.BasicMonodomainSolver  
attribute), 31

time() (cbcbeat.cardiacmodels.CardiacModel method),  
24

## V

values() (cbcbeat.markerwisefield.Markerwise method),  
29

variational\_forms() (cbcbeat.bidomainsolver.BidomainSolver  
method), 23

variational\_forms() (cbcbeat.monodomainsolver.MonodomainSolver  
method), 32