cbcbeat Documentation

Release 1.0

cbcbeat-authors

Mar 29, 2019

Contents

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 assimillation 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:

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 cbcbeat 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.

CHAPTER 6

Documentation:

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=N*
init_condit

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`**(*para*
*init_*

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_condition

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:

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

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

where

$$I_{ion}(v, s) = gv(1 - v/v_th)(1 - v/v_p) + \eta_1 vs$$
$$F(v, s) = \eta_2(v/vp - \eta_3 s)$$

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.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_condition
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_condition
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

$$v_t - \mathrm{div}(G_i v + G_i u) = I_s$$
$$\mathrm{div}(G_i v + (G_i + G_e) u) = I_a$$

where the subscript t denotes the time derivative; G_x denotes a weighted gradient: $G_x = M_x \mathrm{grad}(v)$ 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_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(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_n*)

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

Arguments

k_n (`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_i*, *M_e*, *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_i (`ufl.Expr`) the intra-cellular conductivity as an ufl Expression

M_e (`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_s 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_i, M_e) (`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 `dolfin.GenericFunction` or a Markerwise. NB: it is assumed that the time dependence of I_s 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 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 `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

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_s (`dolfin.Expression`, optional) A typically time-dependent external stimulus. NB: it is assumed that the time dependence of I_s 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.gotran2dolfin module

class `cbcbeat.gotran2dolfin.`**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

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

where the subscript t denotes the time derivative; G_i denotes a weighted gradient: $G_i = M_i \mathrm{grad}(v)$ for, where M_i is the intracellular cardiac conductivity tensor; I_s ise 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_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(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 v_ is in the correct state for t0, gives self.v in correct state at t1.

variational_forms(*k_n*)

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

Arguments

k_n (`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

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

where

- the subscript t denotes the time derivative,

- M_i and M_e are conductivity tensors (in mm^2/ms)
- I_s is prescribed input current (in mV/ms)
- I_a is prescribed input current (in mV/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

- χ is the surface-to volume ratio of cells (in 1/mm) ,
- C_m is the specific membrane capacitance (in mu F/(mm^2)),

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

$$v(x, 0) = v_0$$
$$s(x, 0) = s_0$$

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 interchangably. 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 (`dolfin.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" (`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.

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

Python Module Index

Index