Electrophysiology Module v. 1.0 documentation

Simone Rossi

April 24, 2014

Contents

Chapter 1

Introduction

This document represent a short introduction to the electrophysiology module in LifeV. At the moment the module includes only the monodomain solver which can be coupled with several ionic models. The solver uses expression template assembly in order to assemble the required matrices.

The monodomain solver (ElectroETAMonodomainSolver.hpp) is template on the ionic models. In particular, the abstract base class of the ionic models (ElectroIonicModel.hpp) implements the required assembly of the ionic current in a general form (but without using expression template assembly). This means that if we are to implement a new ionic model, we will have to introduce only the definition of the ionic model (as usually done in the electrophysiology community) in a zero-dimensional form, and the base class will take care of the assembly in three dimensions. This is of great help, especially for biophysically detailed models with many equations, where one can focus on the development of ionic model as a separate entity.

The available ionic models are

- Aliev-Panfilov model (2 variables)
- FitzHugh-Nagumo model (2 variables)
- Fox model (13 variables)
- Goldbeter model, minimal model for slow Calcium waves (2 variables)
- Hodgkin-Huxley model (4 variables)
- Luo-Rudy Phase I model (8 variables)
- Minimal model (Bueno-Orovio et al.) (4 variables)
- Mitchell-Schaeffer model (2 variables)
- Noble model for Purkinje fibers (4 variables)
- Ten Tusscher model second version, 2006 (19 variables)

The testsuite of the present module test the all the above ionic models in their 0D version. All of them are also working in 3D.

The monodomain can be solved using 4 different methods (see the test_benchmark for the implementation):

- Operator Splitting
- Ionic Current Interpolation (ICI)
- Lumped Ionic Current Interpolation (L-ICI, sometimes referred to as Half-Lumping)
- State Variable Interpolation (SVI)

At the moment, the first order operator splitting method is fully tested and allows for subiterations on the reaction part. A second order splitting scheme is available in the solver, but it is still in experimental form. Note that the operator splitting method without subiterations and L-ICI are the same method when lumping the mass matrix. We strongly recommend the lumping of the mass matrix in front of the time dependent term as it "controls" the undershoots that may lead to possible instabilities.

The module comes with 18 tests and 3 examples. The example represent non maintained code, that I thought it may be interested to keep for future references.

In this notes, I will briefly introduce the monodomain model and its numerical approximation by finite element. Then a "tutorial" will introduce you to LifeV and in particular to the electrophysiology module in LifeV. Eventually I will briefly describe each of the tests provided in the testsuite.

1.1 Desirable features for the next releases

Here is a list of desirable features for future releases.

Short term goals (implementable in less than 1 month (by "experienced" LifeV developers))

- O'Hara - Rudy human ionic model
- Bidomain solver (please use composition with the monodomain solver)
- Pseudo-ECG solver
- Optogenetic stimulus
- Use of elementwise parameters to define infarcted areas
- Set orthotropic conductivities
- Better testing of pacing protocols

Long term goals (unknown implementation time)

- Development of a library of ionic models interfacing with cellml
- Development of Purkinje trees solver (currently under development)
- Development of physical-based preconditioners for the bidomain model
- Devolempent of an implicit solver for the mono/bidomain
- Development of a solver for the solution of the eikonal equation
- Development of high-order methods
- Multiscale models for electrophysiology
- Fractional laplacian models.

Chapter 2

Mathematical Overview

We are interested in simulating the propagation of a traveling pulse in biological tissue. At the moment the electrophysiology module is meant for heart applications and therefore the available implemented models consider cardiac cells.

2.1 Ionic models

An ionic model describes the evolution of the action potential on a single cell. It's a set of ordinary differential equations modeling the evolution of the transmembrane electric potential and ionic currents. From a circuital model of the cell we find

$$C_m \frac{\partial V}{\partial t} + \sum_j I_j = I_{app},$$

where $C_m$is the membrane capacitance, V is the transmembrane potential, I_j is the j-th ionic current (density) and I_{app} is the applied current (density). The ionic model specifies how to compute all the needed I_j. Usually a linear relation is assumed for the ionic currents such that the ionic current I_S relative to the ionic species S is

$$I_S = g_S \left(V - E_S\right),$$

where E_S is the resting potential relative to S and g_S is the conductance of the ionic channel. The conductance is a function such that

$$g_S = g_S^{MAX} \phi \left(V, t\right),$$

with $\phi \in [0, 1]$. As an example, in the Hodkin-Huxley ionic model the potassium current is

$$I_K = g_K^{MAX} n^4 \left(V - E_K\right),$$

where $\phi \left(V, t\right) = n = n \left(V, t\right)$. The variable n is called gating variable as it describes the opening and closing of the potassium channel. For $n = 0$, the potassium channel is closed and the potassium current is null. As $n \to 1$, then $I_K \to g_K^{MAX} \left(V - E_K\right)$. Consider the generic gating variable w. It's evolution is described by the equation

$$\frac{\partial w}{\partial t} = \alpha \left(V\right) \left(1 - w\right) - \beta \left(V\right) w.$$

The opening and closing of the ionic channel can be described by several gating variables, such that

$$I_S = g_S^{MAX} w_1^{P_1^S} w_2^{P_2^S} \ldots w_N^{P_N^S} \left(V - E_S\right).$$

We denote by $\mathbf{w}$ the vector of gating variables. Moreover, the resting potential depends on the intracellular and extracellular concentration of the ionic species. Therefore, some ionic model also describe the evolution of the

intracellular ionic concentrations. We denote by $\mathbf{c}$ the vector of ionic concentrations. The full ionic model reads

$$\begin{aligned} C_m \frac{\partial V}{\partial t} + \sum_j I_j &= I_{app}, \\ \frac{\partial \mathbf{w}}{\partial t} &= R\left(V, \mathbf{w}, t\right), \\ \frac{\partial \mathbf{c}}{\partial t} &= S\left(V, \mathbf{w}, \mathbf{c}, t\right), \end{aligned}$$

with some initial conditions $V\left(0\right) = V_0$, $\mathbf{w}\left(0\right) = \mathbf{w}_0$ and $\mathbf{c}\left(0\right) = \mathbf{c}_0$.

Note that not all ionic model use gating variables. For example, simplified two-variables models use a repolarization variable, while some recent biophysically detailed models use Markov chains.

2.2 Monodomain model

The monodomain model can be rigorously derived from the bidomain model. Refer to [7] for details.

The action potential spreads in the tissue as a traveling pulse. We introduce spatial dependence in the ionic model using a diffusion operator such that

$$C_m \frac{\partial V}{\partial t} + \sum_j I_j = \nabla \cdot \left(\frac{1}{\chi} \mathbf{D} \nabla V \right) + I_{app},$$

where χ is the cellular surface-to-volume ratio and $\mathbf{D}$ is the conductivity tensor. Typically anisotropic conductivity is considered such that

$$\mathbf{D} = \sigma_t \mathbf{I} + \left(\sigma_f - \sigma_t\right) \mathbf{f} \otimes \mathbf{f},$$

where $\mathbf{f}$ is the fiber direction, σ_f is the fiber conductivity and σ_t is the cross-fiber conductivity. Insulating boundary conditions are usually imposed such that

$$\left(\mathbf{D}_m \nabla V\right) \cdot \boldsymbol{\nu} = 0,$$

where $\boldsymbol{\nu}$ is the surface normal.

Chapter 3

Numerical Discritization

3.1 Monodomain model

3.1.1 Weak formulation

Consider the monodomain model in the domain Ω:

$$\begin{cases} \chi\left[C_m\dfrac{\partial V}{\partial t}+I_{ion}\left(V,\mathbf{w},\mathbf{c}\right)\right]=\nabla\cdot\left(\mathbf{D}_m\nabla V\right)+I_{app}^m & \text{in } \Omega\times(0,T) \\ \dfrac{\partial \mathbf{w}}{\partial t}=\mathbf{R}\left(V,\mathbf{w}\right), \qquad \dfrac{\partial \mathbf{c}}{\partial t}=\mathbf{S}\left(V,\mathbf{w},\mathbf{c}\right) & \text{in } \Omega\times(0,T) \\ \left(\mathbf{D}_m\nabla V\right)\cdot\boldsymbol{\nu}=0 & \text{in } \partial\Omega\times(0,T) \\ V\left(t_0\right)=V_0, \quad \mathbf{w}\left(t_0\right)=\mathbf{w}_0, \quad \mathbf{c}\left(t_0\right)=\mathbf{c}_0 & \text{in } \Omega. \end{cases} \tag{3.1}$$

The weak formulation of (3.1) reads[3]: given V_0, $\mathbf{w}_0$ and $\mathbf{c}_0$, find the triplet $(V,\mathbf{w},\mathbf{c})\in L^2\left(0,T;W\right)\times L^2\left(0,T;Q^M\right)\times L^2\left(0,T;Q^P\right)$ such that

$$\begin{aligned} \int_\Omega C_m\frac{\partial V}{\partial t}\hat{v}\ \mathrm{d}V+\int_\Omega\frac{1}{\chi}\mathbf{D}_m\nabla V\cdot\nabla\hat{v}\ \mathrm{d}V &= \int_\Omega\left[I_{app}^m-I_{ion}\left(V,\mathbf{w}\right)\right]\hat{v}\ \mathrm{d}V \qquad \forall\hat{v}\in W, \\ \int_\Omega\frac{\partial \mathbf{w}}{\partial t}\cdot\hat{\mathbf{w}}\ \mathrm{d}V &= \int_\Omega\mathbf{R}\left(V,\mathbf{w}\right)\cdot\hat{\mathbf{w}}\ \mathrm{d}V \qquad \forall\hat{\mathbf{w}}\in Q^M, \\ \int_\Omega\frac{\partial \mathbf{c}}{\partial t}\cdot\hat{\mathbf{c}}\ \mathrm{d}V &= \int_\Omega\mathbf{S}\left(V,\mathbf{w},\mathbf{c}\right)\cdot\hat{\mathbf{c}}\ \mathrm{d}V \qquad \forall\hat{\mathbf{c}}\in Q^P, \end{aligned} \tag{3.2}$$

where M and P are the size of the gating variables vector $\mathbf{w}$ and concentration vector $\mathbf{c}$, respectively, and $W=H^1\left(\Omega\right)$, $Q=L^2\left(\Omega\right)$. Introducing the bilinear forms

$$m\left(\mathbf{u},\hat{\mathbf{u}}\right)=\int_\Omega\frac{\partial \mathbf{u}}{\partial t}\cdot\hat{\mathbf{u}}\ \mathrm{d}V,\ s\left(u,\hat{u}\right)=\int_\Omega\frac{1}{\chi}\mathbf{D}_m\nabla u\cdot\nabla\hat{u}\ \mathrm{d}V,$$

and the functionals

$$\begin{aligned} I\left(V,\mathbf{w},\hat{v}\right) &= \int_\Omega\left[I_{app}^m-I_{ion}\left(V,\mathbf{w}\right)\right]\hat{v}\ \mathrm{d}V, \\ R\left(V,\mathbf{w},\hat{\mathbf{w}}\right) &= \int_\Omega\mathbf{R}\left(V,\mathbf{w}\right)\cdot\hat{\mathbf{w}}\ \mathrm{d}V, \\ S\left(V,\mathbf{w},\mathbf{c},\hat{\mathbf{c}}\right) &= \int_\Omega\mathbf{S}\left(V,\mathbf{w},\mathbf{c}\right)\cdot\hat{\mathbf{c}}\ \mathrm{d}V, \end{aligned}$$

we write (3.1) as: given V_0, $\mathbf{w}_0$ and $\mathbf{c}_0$, find the triplet $(V,\mathbf{w},\mathbf{c})\in L^2\left(0,T;W\right)\times L^2\left(0,T;Q^M\right)\times L^2\left(0,T;Q^P\right)$ such that

$$\begin{aligned} m\left(C_mV,\hat{v}\right)+s\left(V,\hat{v}\right)=I\left(V,\mathbf{w},\hat{v}\right) \qquad & \forall\hat{v}\in W, \\ m\left(\mathbf{w},\hat{\mathbf{w}}\right)=R\left(V,\mathbf{w},\hat{\mathbf{w}}\right) \qquad & \forall\hat{\mathbf{w}}\in Q^M, \\ m\left(\mathbf{c},\hat{\mathbf{c}}\right)=S\left(V,\mathbf{w},\mathbf{c},\hat{\mathbf{c}}\right) \qquad & \forall\hat{\mathbf{c}}\in Q^P. \end{aligned} \tag{3.3}$$

Introducing the finite dimensional spaces $W^h \subset W$ and $Q^h \subset Q$, with $\dim W^h = \dim Q^h = N < \infty$, the Galerkin approximation is: given V_0, $\mathbf{w}_0$ and $\mathbf{c}_0$, for every $t \in (0,T)$, find $(V_h, \mathbf{w}_h, \mathbf{c}_h) \in W^h \times \left[Q^h\right]^M \times \left[Q^h\right]^P$ such that

$$\begin{aligned} m\left(C_m V_h, \hat{v}_h\right) + s\left(V_h, \hat{v}_h\right) &= I\left(V_h, \mathbf{w}_h, \hat{v}_h\right) && \forall \hat{v}_h \in W^h, \\ m\left(\mathbf{w}_h, \hat{\mathbf{w}}_h\right) &= R\left(V_h, \mathbf{w}_h, \hat{\mathbf{w}}_h\right) && \forall \hat{\mathbf{w}}_h \in \left[Q^h\right]^M, \\ m\left(\mathbf{c}_h, \hat{\mathbf{c}}_h\right) &= S\left(V_h, \mathbf{w}_h, \mathbf{c}_h, \hat{\mathbf{c}}_h\right) && \forall \hat{\mathbf{c}}_h \in \left[Q^h\right]^P. \end{aligned} \tag{3.4}$$

Next, now introducing the basis $\{\varphi_i\}_{i=1}^N$, $\varphi_i \in W^h$, and $\{\phi_i\}_{i=1}^N$, $\phi_i \in Q^h$, and expand $V_h = V_i\varphi_i$, $\mathbf{w}_h = w_i\boldsymbol{\phi}_i^M$ and $\mathbf{c}_h = c_i\boldsymbol{\phi}_i^P$, where $\boldsymbol{\phi}_i^M$ and $\boldsymbol{\phi}_i^P$ are the basis of the spaces $\left[Q^h\right]^M$ and $\left[Q^h\right]^P$, such that

$$\begin{aligned} C_m \frac{\partial V_i}{\partial t} m\left(\varphi_i, \varphi_j\right) + V_i s\left(\varphi_i, \varphi_j\right) &= I\left(V_i\varphi_i, w_i\boldsymbol{\phi}_i^M, c_i\boldsymbol{\phi}_i^P; \varphi_j\right) && \forall j = 1, ..., N, \\ w_i m\left(\boldsymbol{\phi}_i^M, \boldsymbol{\phi}_j^M\right) &= R\left(V_i\varphi_i, w_i\boldsymbol{\phi}_i^M; \boldsymbol{\phi}_j^M\right) && \forall j = 1, ..., NM, \\ c_i m\left(\boldsymbol{\phi}_i^P, \boldsymbol{\phi}_j^P\right) &= S\left(V_i\varphi_i, w_i\boldsymbol{\phi}_i^M, c_i\boldsymbol{\phi}_i^P; \boldsymbol{\phi}_j^P\right) && \forall j = 1, ..., NP. \end{aligned} \tag{3.5}$$

Denoting by $\mathbb{M}_{ji} = m\left(\varphi_i, \varphi_j\right)$ and $\mathbb{M}_{ji}^K = m\left(\boldsymbol{\phi}_i^K, \boldsymbol{\phi}_j^K\right)$, and $\mathbb{K}_{ji} = s\left(\varphi_i, \varphi_j\right)$ we can write the following semidiscrete algebraic problem

$$\begin{vmatrix} C_m\mathbb{M} & 0 & 0 \\ 0 & \mathbb{M}^M & 0 \\ 0 & 0 & \mathbb{M}^N \end{vmatrix} \begin{vmatrix} \dot{\mathbb{V}} \\ \dot{\mathbb{W}} \\ \dot{\mathbb{C}} \end{vmatrix} = \begin{vmatrix} \mathbb{I} - \mathbb{K}\mathbb{V} \\ \mathbb{R} \\ \mathbb{S} \end{vmatrix}, \tag{3.6}$$

where the coupling between the fields is encoded on the right hand side. Following [11], we assume that there exists a weak coupling between the gating variables $\mathbf{w}$, the ionic concentrations $\mathbf{c}$ and the potential V. This strategy is motivated by the fact that the Jacobian of the full implicit system is dominated by diagonal entries. Therefore, for a sufficiently small time interval, we have the three separate problems:

$$C_m\mathbb{M}\dot{\mathbb{V}} + \mathbb{K}\mathbb{V} = \mathbb{I}, \quad \mathbb{M}^M\dot{\mathbb{W}} = \mathbb{R}, \quad \mathbb{M}^N\dot{\mathbb{C}} = \mathbb{S}.$$

3.1.2 Fully discrete approximation

The vectors $\mathbb{R}$ and $\mathbb{S}$ are commonly approximated by

$$\mathbb{R} = \mathbb{M}^M\widetilde{\mathbb{R}}, \quad \mathbb{S} = \mathbb{M}^N\widetilde{\mathbb{S}},$$

where $\widetilde{\mathbb{R}}$ and $\widetilde{\mathbb{S}}$ represent the nodal reaction terms. In this way the gating variables and the concentration equations can be solved at each node independently, that is, the semidiscrete problems $\mathbb{M}^M\dot{\mathbb{W}} = \mathbb{R}$ and $\mathbb{M}^N\dot{\mathbb{C}} = \mathbb{S}$ are equivalent to the node-wise gating variables and concentration ODEs.

Discretizing in time, such that $t^n = t^0 + n\Delta t$, we denote by $\mathbb{V}^n = \mathbb{V}\left(t^n\right)$, $\mathbb{W}^n = \mathbb{W}\left(t^n\right)$ and $\mathbb{C}^n = \mathbb{C}\left(t^n\right)$. It is possible to take advantage of it using a Rush-Larsen scheme [16] of the type

$$\mathbb{W}^{n+1} = \mathbb{W}_\infty\left(\mathbb{V}^n\right) + \left[\mathbb{W}^n - \mathbb{W}_\infty\left(\mathbb{V}^n\right)\right] e^{-\Delta t/\tau_{\mathbb{W}}}.$$

The concentration equations are usually solved by the forward Euler scheme

$$\mathbb{C}^{n+1} = \mathbb{C}^n + \Delta t\widetilde{\mathbb{S}}\left(\mathbb{V}^n, \mathbb{W}^n, \mathbb{C}^n\right).$$

Several approximations have been used to solve the reaction-diffusion equation $C_m\mathbb{M}\dot{\mathbb{V}} + \mathbb{K}\mathbb{V} = \mathbb{I}$. The most common approach is the operator splitting method, such that, for a first order splitting,

$$\begin{aligned} C_m\mathbb{M}\dot{\mathbb{V}}_1 = \mathbb{I}\left(\mathbb{V}_1\right), &\quad \text{for } t^n \le t \le t^{n+1}, \quad \text{with } \mathbb{V}_1^n = \mathbb{V}^n, \\ C_m\mathbb{M}\dot{\mathbb{V}}_2 + \mathbb{K}\mathbb{V}_2 = 0, &\quad \text{for } t^n \le t \le t^{n+1}, \quad \text{with } \mathbb{V}_2^n = \mathbb{V}_1^{n+1}, \\ \mathbb{V}^{n+1} = \mathbb{V}_2^{n+1}. & \end{aligned}$$

while for a second order splitting

$$\begin{aligned} C_m\mathbb{M}\dot{\mathbb{V}}_1 = \mathbb{I}\left(\mathbb{V}_1\right), &\quad \text{for } t^n \le t \le t^{n+1/2}, \quad \text{with } \mathbb{V}_1^n = \mathbb{V}^n, \\ C_m\mathbb{M}\dot{\mathbb{V}}_2 + \mathbb{K}\mathbb{V}_2 = 0, &\quad \text{for } t^n \le t \le t^{n+1}. \quad \text{with } \mathbb{V}_2^n = \mathbb{V}_1^{n+1/2}, \\ C_m\mathbb{M}\dot{\mathbb{V}}_3 = \mathbb{I}\left(\mathbb{V}_3\right), &\quad \text{for } t^{n+1/2} \le t \le t^{n+1}, \quad \text{with } \mathbb{V}_3^n = \mathbb{V}_2^{n+1}, \\ \mathbb{V}^{n+1} = \mathbb{V}_3^{n+1}. & \end{aligned}$$

In view of parallel computations, the reaction step can be solved independently on each processor. In this case, as for the gating variables, the right hand side is typically approximated as

$$\mathbb{I} = \mathbb{M}\widetilde{\mathbb{I}}. \tag{3.7}$$

Approximation (3.7) is usually called ionic current interpolation (ICI). Although several methods can be used to solve the reaction step, we will only consider the forward Euler case. In this case, the discrete system reads

$$\begin{aligned} \mathbb{V}_1^{n+q/K} = \mathbb{V}_1^{n+(q-1)/K} + \frac{\Delta t}{C_m K}\widetilde{\mathbb{I}}^{n+(q-1)/K}, \quad \text{for } q = 1, ..., K \quad \text{with } \mathbb{V}_1^n = \mathbb{V}^n, \\ \left(\frac{C_m}{\Delta t}\mathbb{M} + \mathbb{K}\right)\mathbb{V}_2^{n+1} = \frac{C_m}{\Delta t}\mathbb{M}\mathbb{V}_2^n, \qquad \text{with } \mathbb{V}_2^n = \mathbb{V}_1^{n+1}, \\ \mathbb{V}^{n+1} = \mathbb{V}_2^{n+1}, \end{aligned}$$

where K is the number of subiterations taken in the reaction step.

The other typically preferred time discretization of the monodomain equation is a semi-implicit scheme, such as

$$\left(\frac{C_m}{\Delta t}\mathbb{M} + \mathbb{K}\right)\mathbb{V}^{n+1} = \frac{C_m}{\Delta t}\mathbb{M}\mathbb{V}^n + \mathbb{I}^n. \tag{3.8}$$

It is well known that in general (3.8), as well as the diffusion step in the operator splitting scheme, do not satisfy the discrete maximum principle leading to undershoots and overshoots near the front of the traveling pulse. To reduce these instabilities, mass lumping is typically used [17]. Specifically, we will consider lumping obtained by nodal integration. Therefore, (3.8) becomes

$$\left(\frac{C_m}{\Delta t}\mathbb{M}_L + \mathbb{K}\right)\mathbb{V}^{n+1} = \frac{C_m}{\Delta t}\mathbb{M}_L\mathbb{V}^n + \mathbb{I}^n,$$

where $\mathbb{M}_L$ is the lumped mass matrix. For the ionic currents $I(\mathbf{x}) = I(V(\mathbf{x}), \mathbf{w}(\mathbf{x}), \mathbf{c}(\mathbf{x}))$, we consider the following different approximations:

- **ICI** (Half Lumping): the ionic currents are linearly interpolated, that is

$$\mathbb{I}_{ICI}^n = \int_\Omega I_j^n \varphi_j(\mathbf{x})\varphi_j(\mathbf{x}) = \int_\Omega I\left(V_j^n, \mathbf{w}_j^n, \mathbf{c}_j^n\right)\varphi_j(\mathbf{x})\varphi_j(\mathbf{x}) = \mathbb{M}\widetilde{\mathbb{I}}^n.$$

  The ICI monodomain approximation reads

$$\left(\frac{C_m}{\Delta t}\mathbb{M}_L + \mathbb{K}\right)\mathbb{V}^{n+1} = \frac{C_m}{\Delta t}\mathbb{M}_L\mathbb{V}^n + \mathbb{M}\widetilde{\mathbb{I}}^n.$$

- **Lumped ICI** or **L-ICI** (Full Lumping): the ionic currents are interpolated nodally or, equivalently, the mass matrix arising in the ICI method is lumped such that $\mathbb{I}^n = \mathbb{M}_L\widetilde{\mathbb{I}}^n$ and

$$\left(\frac{C_m}{\Delta t}\mathbb{M}_L + \mathbb{K}\right)\mathbb{V}^{n+1} = \frac{C_m}{\Delta t}\mathbb{M}_L\mathbb{V}^n + \mathbb{M}_L\widetilde{\mathbb{I}}^n.$$

- **SVI** (State Variable Interpolation): the transmembrane potential, the gating variables and the ionic concentrations are evaluated in the quadrature nodes and used to evaluate the ionic currents, such that

$$\begin{aligned} V^n(\mathbf{X}) &= V_j^n\varphi_j(\mathbf{X}), \\ \mathbf{w}^n(\mathbf{X}) &= w_j^n\boldsymbol{\phi}_j^M(\mathbf{X}), \\ \mathbf{c}^n(\mathbf{X}) &= c_j^n\boldsymbol{\phi}_j^N(\mathbf{X}), \\ \mathbb{I}_{SVI}^n &= \int_\Omega I\left(V_j^n, \mathbf{w}_j^n, \mathbf{c}_j^n\right)\varphi_j(\mathbf{x}) = \int_\Omega I\left(V_j^n\varphi_j, w_j^n\boldsymbol{\phi}_j^M, c_j^n\boldsymbol{\phi}_j^N\right)\varphi_j(\mathbf{x}), \end{aligned}$$

  and

$$\left(\frac{C_m}{\Delta t}\mathbb{M}_L + \mathbb{K}\right)\mathbb{V}^{n+1} = \frac{C_m}{\Delta t}\mathbb{M}_L\mathbb{V}^n + \mathbb{I}_{SVI}^n.$$

While several studies have focused on the different methods [14] and on studying different lumping strategies [8], we believe that lumping must be enforced to reduce instabilities, but the approximation obtained by a full lumping scheme is not accurate enough, as we will see in the following chapter. Moreover, the first order operator splitting method with only one subiteration in the reaction step is equivalent to the full lumping scheme. In fact, since $\mathbb{V}^{n+1} = \mathbb{V}_2^{n+1}$, then

$$\left(\frac{C_m}{\Delta t}\mathbb{M}_L + \mathbb{K}\right)\mathbb{V}^{n+1} = \left(\frac{C_m}{\Delta t}\mathbb{M}_L + \mathbb{K}\right)\mathbb{V}_2^{n+1} = \frac{C_m}{\Delta t}\mathbb{M}_L\mathbb{V}_1^{n+1} = \frac{C_m}{\Delta t}\mathbb{M}_L\mathbb{V}^n + \mathbb{M}_L\widetilde{\mathbb{I}}^n.$$

We consider $\mathbb{P}_1$ elements for all electrophysiologically related fields.

Chapter 4

Tutorial

To start using the electrophysiology module, the suggested starting point consists in implementing an ionic model. There are many ionic models in the literature, just pick one that you believe it may be useful for you. I will show you how to introduce a new ionic model in LifeV and how to make a simple test out of it. In the third tutorial we will introduce the model to a 3D simulation. Eventually I will show you how I build the fiber and sheet directions on ventricular geometries.

I assume you may not be experienced C++ developer, but at the same time this is not an introduction to C++. Here and there, I will introduce you to some common LifeV developing rules. More details can be found in the LifeV developers guidelines.

4.1 Introducing a new ionic model

To show how to introduce a new ionic model we consider the Hodgkin-Huxley model. The model equations can be found in [7]. The model has three gating variables, M, N and H and no ionic concentration. In any case, I would recommend to check out the implementation of the other ionic models, to get more familiar on how these are implemented.

Let's create the files IonicHodgkinHuxley.hpp and IonicHodgkinHuxley.cpp. In the hpp file we need to include the base ionic model class, and Teuchos parameter list to import parameters from an xml file. Teuchos is a Trilinos package. Make sure you have installed it correctly.

```
//IonicHodgkinHuxley.hpp

//Include the base class
#include <lifev/electrophysiology/solver/IonicModels/ElectroIonicModel.hpp>

//include Teuchos for importing parameters from an xml file
#include <Teuchos_ParameterList.hpp>

namespace LifeV {
//! IonicModel - This class implements the Hodgkin-Huxley ionic model.

class IonicHodgkinHuxley : public virtual ElectroIonicModel
{
        /* Hodgkin Huxley class implementation here */
}

}// end LifeV namespace
```

Note that the above class inherits from the ElectroIonicModel class. The inheritance should be kept as virtual to allow for multiple inheritance.

The structure of the class is as following: at the beginning we have typedef, then the constructors and the desctructor, afterward setters and getters, followed by other methods and evetually the private members

```
//IonicHodgkinHuxley.hpp

class IonicHodgkinHuxley : public virtual ElectroIonicModel
{
public:
    //! @name Type definitions
    //@{
    // typedef here ...
    //@}

    //! @name Constructors & Destructor & Overloads
    //@{
    // Constructor and destructor here ...
    //@}

    //! @name Setters and getters
    //@{
    // getters ...
    // setters ...
    //@}

    //! @name Methods
    //@{
    // Other methods ...
    //@}

private:
    //! @name Private members
    //@{
    // Model parameters ...
    //@}
}
```

The HodgkinHuxley class member are the parameters of the model. In LifeV, the members of a class have the prefix "M_". Also not that the members are define as "Real". The Real type is equivalent to a "double". All members should be private and accessible through public getters and setters. We report here just an example of a getter and of a setter for one of the private members. The getter has two const keyword. The first tells you that the value you will get cannot be changed, while the second is telling you that the method is not changing any member of the class. Moreover, we "inline" the methods when they are very short (such as getters and setters).

Recall that according to the LifeV guidelines,

- Member variables should have the prefix M_: e.g. M_variable;
- The getter should not be preceded by the word "get": e.g. variable();
- The setter should be preceded by the word set, and when combining more words, you should not use underscores but use capitals; e.g. setVariable(var);

```
//IonicHodgkinHuxley.hpp

class IonicHodgkinHuxley : public virtual ElectroIonicModel
{
public:
//! @name Type definitions
    //@{
    // typedef here ...
    //@}
```

```
    //! @name Constructors & Destructor & Overloads
    //@{
    // Constructor and destructor here ...
    //@}

    //! @name Setters and getters
    //@{
    //parameters getters and setters
    inline const Real& gNa() const
    {
        return M_gNa;
    }

    // Other getters ...

    inline void setGNa ( const Real& p )
    {
        M_gNa = p;
    }

    // Other setters ...
    //@}

    //! @name Methods
    //@{
    // Other methods ...
    //@}

private:
    //! @name Private members
    //@{
    //! Model Parameters
    //! Chemical kinetics parameters
    Real M_gNa;    //Maximal conducntunce for Na currents
    Real M_gK;     //Maximal conducntunce for K  currents
    Real M_gL;     //Maximal conducntunce for Cl currents
    Real M_vNa;    //Resting potential for Na
    Real M_vK;     //Resting potential for K
    Real M_vL;     //Resting potential for Cl
    //@}
}
```

Now we sprecify the typedefs and the constructor. We call the base class with "keyword" super. We define:

- an empty constructor where we will set up the model with default parameters
- a constructor with a Teuchos::ParameterList as argument. The parameters should be written in an xml file, read and passed to the constructor. See later for more information.
- a copy constructor
- a virtual destructor
- an operator assignement

```
//IonicHodgkinHuxley.hpp

class IonicHodgkinHuxley : public virtual ElectroIonicModel
{
```

```
public:
    //! @name Type definitions
    //@{
    typedef ElectroIonicModel                          super;
    //@}

    //! Empty constructor
    IonicHodgkinHuxley();

    //! Empty constructor
    /*!
     * @param parameterList list of parameters in an xml file
     */
    IonicHodgkinHuxley ( Teuchos::ParameterList& parameterList );

    //! Empty constructor
    /*!
     * @param IonicHodgkinHuxley object
     */
    IonicHodgkinHuxley ( const IonicHodgkinHuxley& model );

    //! Destructor
    virtual ~IonicHodgkinHuxley() {}

    //@}

    //! @name Overloads
    //@{
    IonicHodgkinHuxley& operator= ( const IonicHodgkinHuxley& model );
    //@}

    //! @name Setters and getters
    //@{
    // getters ...
    // setters ...
    //@}

    //! @name Methods
    //@{
    // Other methods ...
    //@}

private:
    //! @name Private members
    //@{
    // Model parameters ...
    //@}
}
```

Now I will show you the implementation of the above constructors that can be found in the cpp file. Let's start with the empty constructor:

```
//IonicHodgkinHuxley.cpp

IonicHodgkinHuxley::IonicHodgkinHuxley() :
    super( 4, 3 ),
    M_gNa (120.0),
```

```
    M_gK (36.0),
    M_gL (0.3),
    M_vNa (115.0),
    M_vK (-12.0),
    M_vL (10.6)
{
    M_restingConditions.at (0) = 0.0;                   //potential V
    M_restingConditions.at (1) = 0.052932485257250;     //gating variable M
    M_restingConditions.at (2) = 0.317676914060697;     //gating variable N
    M_restingConditions.at (3) = 0.596120753508460;     //gating variable H
}
```

The base class constructor is called through the call `super(4, 3)`. The number 4 represents the number of variable in the model, while 3 is the number of gating variables. The constructor of the base class will create a `std::vector<Real> M_resting` condition with the size equal to the number of equation in the ionic model. We assign to each element of this vector the resting value of the corresponding variable. The transmembrane potential should always be at position 0. After the potential I list the gating variables. After the gating variable all the other variables (not present in this case). The resting values are important to initialize all the simulations in an easy way. The resting conditions can be changed by using the setters of the ElectroIonicModel class.

```
//IonicHodgkinHuxley.cpp

IonicHodgkinHuxley::IonicHodgkinHuxley ( Teuchos::ParameterList& parameterList ) :
    super( 4, 3 )
{
    M_gNa =  parameterList.get ("gNa", 120.0 );
    M_gK  =  parameterList.get ("gK" ,  36.0 );
    M_gL  =  parameterList.get ("gL" ,   0.3 );
    M_vNa =  parameterList.get ("vNa", 115.0 );
    M_vK  =  parameterList.get ("vK" , -12.0 );
    M_vL  =  parameterList.get ("vL" ,  10.6 );

    M_restingConditions.at (0) = 0.0;                   //potential V
    M_restingConditions.at (1) = 0.052932485257250;     //gating variable M
    M_restingConditions.at (2) = 0.317676914060697;     //gating variable N
    M_restingConditions.at (3) = 0.596120753508460;     //gating variable H
}
```

The parameters are accessible through the method get. The string between quotes refers to the name of the variable in the xml file, and the following number represent a default value to be used if the parameter has not been specified. For example the xml should look like this:

```
//HodgkinHuxleyParameters.xml

<!-- Model Parameters -->
    <ParameterList name="Hodgkin Huxley: Parameter list">
    <Parameter name="gNa" type="double" value="120.0"/>
    <Parameter name="gK"  type="double" value="36.0" />
    <Parameter name="gL"  type="double" value="0.3"  />
    <Parameter name="vNa" type="double" value="115.0"/>
    <Parameter name="vK"  type="double" value="-12.0"/>
    <Parameter name="vL"  type="double" value="10.6" />
    </ParameterList>
```

The copy constructor and the assignment operator look like the following

```
//IonicHodgkinHuxley.cpp

//! Copy constructor
```

```
IonicHodgkinHuxley::IonicHodgkinHuxley ( const IonicHodgkinHuxley& model )
{
    M_gNa =  model.M_gNa;
    M_gK  =  model.M_gK;
    M_gL  =  model.M_gL;
    M_vNa =  model.M_vNa;
    M_vK  =  model.M_vK;
    M_vL  =  model.M_vL;

    M_numberOfEquations = model.M_numberOfEquations;
    M_restingConditions = model.M_restingConditions;
    M_numberOfGatingVariables = model.M_numberOfGatingVariables;
}

//! Assignment operator
IonicHodgkinHuxley& IonicHodgkinHuxley::operator= ( const IonicHodgkinHuxley& model )
{
    M_gNa =  model.M_gNa;
    M_gK  =  model.M_gK;
    M_gL  =  model.M_gL;
    M_vNa =  model.M_vNa;
    M_vK  =  model.M_vK;
    M_vL  =  model.M_vL;

    M_numberOfEquations = model.M_numberOfEquations;
    M_restingConditions = model.M_restingConditions;
    M_numberOfGatingVariables = model.M_numberOfGatingVariables;

    return      *this;
}
```

Finally we need to implement the equation of the ionic model. In order to do this, we need to overload the following abstract methods of the base class

- `void computeGatingRhs ( const std::vector<Real>& v, std::vector<Real>& rhs);`
- `Real computeLocalPotentialRhs ( const std::vector<Real>& v);`
- `void computeRhs ( const std::vector<Real>& v, std::vector<Real>& rhs);`
- `void showMe();`

Moreover, we may want to solve the gating variables with Rush-Larsen scheme. If needed you can overload also the following method

- `void computeGatingVariablesWithRushLarsen ( std::vector<Real>& v, const Real dt);`

The above methods are automatically used in the assembly of the ionic current vector in 3D simulations.

```
//IonicHodgkinHuxley.hpp

class IonicHodgkinHuxley : public virtual ElectroIonicModel
{
public:
    //! @name Type definitions
    //@{
    // typedef here ...
    //@}

    //! @name Constructors & Destructor & Overloads
```

```
    //@{
    // Constructor and destructor here ...
    //@}

    //! @name Setters and getters
    //@{
    // getters ...
    // setters ...
    //@}

    //! @name Methods
    //@{
    //! Compute the rhs of the gating variables a single node or for the 0D case
    /*!
     * @param v vector with the variables (V, M, N, H)
     * @param rhs vector where we will insert the rhs of the equations of the gating variables
     */
    void computeGatingRhs ( const std::vector<Real>& v, std::vector<Real>& rhs);

    //! compute the rhs of the potential equation on a single node or for the 0D case
    /*!
     * @param v vector with the variables (V, M, N, H)
     */
    Real computeLocalPotentialRhs ( const std::vector<Real>& v );

    //! Compute the rhs on a single node or for the 0D case
    /*!
     * @param v vector with the variables (V, M, N, H)
     * @param rhs vector where we will insert the rhs of the equations
     */
    void computeRhs ( const std::vector<Real>& v, std::vector<Real>& rhs);

    //! compute the rhs of the gating variables with the RushLarsen scheme
    /*!
     * @param v vector with the variables (V, M, N, H)
     * @param dt timestep
     */
    void computeGatingVariablesWithRushLarsen ( std::vector<Real>& v, const Real dt );

    //! Display information about the model
    void showMe();

    //@}

private:
    //! @name Private members
    //@{
    // Model parameters ...
    //@}
}
```

In the method `computeGatingRhs`, the vector `v` has size 4, while the vector `rhs` has size `3`. The implementation is as follows

```
//IonicHodgkinHuxley.cpp

void IonicHodgkinHuxley::computeGatingRhs ( const std::vector<Real>&  v, std::vector<Real>& rhs )
```

```
{
    Real V = v[0];
    Real M = v[1];
    Real N = v[2];
    Real H = v[3];

    Real alpham = 0.1 * (25. - V) / (std::exp ( (25. - V) / 10.) - 1.);
    Real betam = 4.*std::exp (-V / 18.0);     Real alphah = 0.07 * std::exp (-V / 20.);
    Real betah = 1.0 / (std::exp ( (30. - V) / 10.) + 1.);
    Real alphan = 0.01 * (10. - V) / (std::exp ( (10. - V) / 10.) - 1.);
    Real betan = 0.125 * std::exp (-V / 80.0);

    rhs[0] = alpham * (1 - M) - betam * M;
    rhs[1] = alphan * (1 - N) - betan * N;
    rhs[2] = alphah * (1 - H) - betah * H;
}
```

The method `computeLocalPotentialRhs` computes the right hand side only of the potential equation. The implementation is as follows

```
//IonicHodgkinHuxley.cpp

Real IonicHodgkinHuxley::computeLocalPotentialRhs ( const std::vector<Real>& v )
{
    Real dPotential (0.0);

    Real V = v[0];
    Real M = v[1];
    Real N = v[2];
    Real H = v[3];

    dPotential = - M_gK * N * N * N * N * (V - M_vK)
                 - M_gNa * M * M * M * H * (V - M_vNa)
                 - M_gL * (V - M_vL);

   return dPotential;
}
```

The method `computeRhs` computed the right hand side of the whole set of equations in the ionic model. I decided to put this method as abstract even if, in principle, this method should call the previous two methods. The main reason is that in some cases it may be more efficient to repeat the whole code. In this case, I decided to show you, how to use the prevoius methods: the first step is to create a temporary right hand side vector to be given to the computeGatingRhs method:

```
//IonicHodgkinHuxley.cpp

void IonicHodgkinHuxley::computeRhs ( const   std::vector<Real>&  v, std::vector<Real>& rhs )
{
    std::vector<Real> tmpRhs(this->Size()-1, 0.0);
    computeGatingRhs(v, tmpRhs);

    rhs[0] = computeLocalPotentialRhs(v);
    rhs[1] = tmpRhs[0];
    rhs[2] = tmpRhs[1];
    rhs[3] = tmpRhs[2];
}
```

Finally, one must implement the `showMe` method. This should show the model parameters, but in some cases they would be too many. In this case, the implementer (thank you Simone!) decided to write:

```
//IonicHodgkinHuxley.cpp

void IonicHodgkinHuxley::showMe()
{
    std::cout << "\n\tHi, I'm the Hodgkin Huxley model for neurons.
                      This is the first model implemented by Simone Palamara!!!\n
                    \t See you soon\n\n";
}
```

The Hodgkin Huxley model can be solved more efficiently using the Rush-Larsen scheme for the gating variables. The method is implemented as follows

```
//IonicHodgkinHuxley.cpp

void IonicHodgkinHuxley::computeGatingVariablesWithRushLarsen ( std::vector<Real>& v, const Real dt )
{
    Real V = v[0];
    Real M = v[1];
    Real N = v[2];
    Real H = v[3];

    Real alpham = 0.1 * (25. - V) / (std::exp ( (25. - V) / 10.) - 1.);
    Real betam = 4.*std::exp (-V / 18.0);
    Real alphah = 0.07 * std::exp (-V / 20.);
    Real betah = 1.0 / (std::exp ( (30. - V) / 10.) + 1.);
    Real alphan = 0.01 * (10. - V) / (std::exp ( (10. - V) / 10.) - 1.);
    Real betan = 0.125 * std::exp (-V / 80.0);

    Real taum = alpham + betam;
    Real taun = alphan + betan;
    Real tauh = alphah + betah;
    Real mInf = alpham / (taum);
    Real nInf = alphan / (taun);
    Real hInf = alphah / (tauh);

    v[1] = mInf + (M - mInf) * std::exp (-dt * taum);
    v[2] = nInf + (N - nInf) * std::exp (-dt * taun);
    v[3] = hInf + (H - hInf) * std::exp (-dt * tauh);
}
```

To compile the file together with the other ionic model we need to include the two file in the CMakeList.txt file

```
#CMakeList.txt

SET( solver_IonicModels_HEADERS
  solver/IonicModels/IonicAlievPanfilov.hpp
  solver/IonicModels/IonicFitzHughNagumo.hpp
  solver/IonicModels/IonicLuoRudyI.hpp
  solver/IonicModels/IonicMitchellSchaeffer.hpp
  solver/IonicModels/IonicTenTusscher06.hpp
  solver/IonicModels/IonicMinimalModel.hpp
  solver/IonicModels/ElectroIonicModel.hpp
  solver/IonicModels/IonicFox.hpp
  solver/IonicModels/IonicHodgkinHuxley.hpp # <-- ADD THIS LINE
  solver/IonicModels/IonicNoblePurkinje.hpp
  solver/IonicModels/IonicGoldbeter.hpp
CACHE INTERNAL "")
```

```
SET( solver_IonicModels_SOURCES
  solver/IonicModels/IonicAlievPanfilov.cpp
  solver/IonicModels/IonicFitzHughNagumo.cpp
  solver/IonicModels/IonicLuoRudyI.cpp
  solver/IonicModels/IonicMitchellSchaeffer.cpp
  solver/IonicModels/IonicTenTusscher06.cpp
  solver/IonicModels/IonicMinimalModel.cpp
  solver/IonicModels/IonicFox.cpp
  solver/IonicModels/ElectroIonicModel.cpp
  solver/IonicModels/IonicHodgkinHuxley.cpp # <-- ADD THIS LINE
  solver/IonicModels/IonicNoblePurkinje.cpp
  solver/IonicModels/IonicGoldbeter.cpp
CACHE INTERNAL "")

SET(LOCAL_HEADERS)
FOREACH(INC ${solver_IonicModels_HEADERS})
 STRING(REPLACE "solver/IonicModels/" "" LOCAL_INC ${INC})
 SET(LOCAL_HEADERS ${LOCAL_HEADERS} ${LOCAL_INC})
ENDFOREACH()

INSTALL(
  FILES ${LOCAL_HEADERS}
  DESTINATION "${${PROJECT_NAME}_INSTALL_INCLUDE_DIR}/lifev/electrophysiology/solver/IonicModels"
  COMPONENT ${PACKAGE_NAME}
)
```

This concludes the tutorial on how to introduce a new ionic model.

4.2 Testing the implementation of the new ionic model

In the second tutorial we use the Hodgkin Huxley model implemented in the previous tutorial and test that it is working. The test can be found in the testsuite folder, in the test_0DHodgkinHuxley.

This is a very simple example with almost no LifeV specific commands. For this reason I will be short.

We start by including the Hodgkin Huxley model we just described. To write the solution we include fstream. Finally we want to work in the LifeV namespace and we need to include LifeV.hpp.

```
//main.cpp

#include <fstream>
#include <lifev/electrophysiology/solver/IonicModels/IonicHodgkinHuxley.hpp>
#include <lifev/core/LifeV.hpp>

using namespace LifeV;
```

After starting the program we read the parameter list xml file using Teuchos.

```
Int main ( Int argc, char** argv )
{
    //*******************************************//
    // Import parameters from an xml list. Use   //
    // Teuchos to create a list from a given file //
    // in the execution directory.               //
    //*******************************************//
    std::cout << "Importing parameters list...";
    Teuchos::ParameterList parameterList =
            * ( Teuchos::getParametersFromXmlFile ( "HodgkinHuxleyParameters.xml" ) );
```

```
    std::cout << " Done!" << std::endl;
```

Now we create the ionic model, and we use it to initialize a vector where we store the solution.

In particular the method Size() tells us the number of equation in the ionic model and is therefore used to create a vector with the correct size. The initialize() method instead fills the elements of the vector "states" with the resting values/initial conditions.

```
// If you do not have the xml parameter list use the empty constructor
IonicHodgkinHuxley  ionicModel(parameterList);
ionicModel.showMe();

std::vector<Real> states (ionicModel.Size(), 0); //Size gives the number of equations of the model
ionicModel.initialize(states);    //fills states with the resting conditions of the model
```

Then we create a vector to store the right hand side of the equations of the ionic model. After we declare the applied current, the final time and the timestep. Before starting to solve we create a file where we can save the solution.

```
std::vector<Real> rhs (ionicModel.Size(), 0); //create a vector for the rhs

Real Iapp (0);    //Value of the applied current
Real TF (40);     //Final time of the simulation
Real dt (0.005);  //timestep

//Create a file named output.txt where we write the solution
std::string filename = "output.txt";
std::ofstream output ("output.txt");
```

We start the time loop. First we update the gating variable using for example the Rush-Larsen method we implemented earlier. Then we compute the sum of the ionic currents of the ionic model. Afterwards, we update the value of the apllied current and therefore the right hand side of the potential equation. We update the value of the potential using forward Euler and the time. Eventually we save the solution on the output file.

```
for ( Real t = 0; t < TF; )
{
    //Use Rush-Larsen to update the gating variables
    ionicModel.computeGatingVariablesWithRushLarsen (states, dt);
    //Compute the value of the potential rhs with the updated gating variables
    Real RHS = ionicModel.computeLocalPotentialRhs ( states);

    //Update the applied current
    if ( t > 20.5 && t < 21 ) Iapp = 380.0;
    else Iapp = 0;
    ionicModel.setAppliedCurrent (Iapp);
    //Update the rhs of the potential equation
    ionicModel.addAppliedCurrent(RHS);
    //solve potential equation using forward Euler
    states[0] = states[0]  + dt * (RHS);

    //update time
    t += dt;

    //Save the solution on the output file
    output << t << ", ";
    for ( int j (0); j < ionicModel.Size() - 1; j++) output << states.at (j) << ", ";
    output << states.at ( ionicModel.Size() - 1 ) << "\n";

}
```

We end the main by closing the output file and finishing the program

```
output.close(); //close output file
return 0;       //return value for main function
}
```

In the folder where we have the main we have just shown, we should also have the xml file with the list of parameter. We can set up a CMakeList.txt file like the following

The name specified in the TRIBITS_ADD_EXECUTABLE is the name of the executable. Actually a prefix with the name of the module will be added as well as the suffix .exe: eventually the executable will have the name Electrophysiology_test_HodgkinHuxley.exe.

In TRIBITS_COPY_FILES_TO_BINARY_DIR instead we create a link between the folder where we have the source files and the executable directory. In particular we link the file HodgkinHuxleyParameters.xml so that any change we will do in that file in the source directory will be immediately available in the executable folder.

```
INCLUDE(TribitsAddExecutableAndTest)
INCLUDE_DIRECTORIES(${CMAKE_SOURCE_DIR})

TRIBITS_ADD_EXECUTABLE(
  test_HodgkinHuxley
  SOURCES main.cpp
)
TRIBITS_COPY_FILES_TO_BINARY_DIR(data_test_HodgkinHuzleyModel_data
  CREATE_SYMLINK
  SOURCE_FILES HodgkinHuxleyParameters.xml
  SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}
)
```

If you would like to create a test, which will be tested together with the testsuite typing "ctest", instead of TRIBITS_ADD_EXECUTABLE we can use TRIBITS_ADD_EXECUTABLE_AND_TEST as follows

```
TRIBITS_ADD_EXECUTABLE_AND_TEST(
  test_HodgkinHuxley
  SOURCES main.cpp
  ARGS -c
  NUM_MPI_PROCS 1
  COMM serial mpi
)
```

If you wish to add this as test, set NUM_MPI_PROCS 1, as there is no need for more processors for this simple simulation.

Remember also that you need to add the folder where you put all these files in the CMakeList.txt of the example/testsuite folder.

4.3 Testing the new ionic model on the electrophysiology benchmark

Once the ionic model has been coded and tested in 0D, we can test it in 3D. If the zero dimensional version is correctly working, the 3D version should work as a black box. On the other hand, notice that the value of the membrane capacitance in the ionic model and the value of the surface to volume ration greatly affects the speed of the travelling pulse. I will start describing the main file in details. If you just wish to introduce your new ionic model, go at the end of this section.

Let's start having a look at the benchmark test. The first header we include is the Monodomain solver. The solver has all the utilities required to solve the monodomain model with the newly developed ionic model.

```
#include <lifev/electrophysiology/solver/ElectroETAMonodomainSolver.hpp>
```

I put some functions used in this test in a separate file in order to keep the main easier to read. We will have a look at this file at the end.

```
#include <lifev/electrophysiology/testsuite/test_benchmark/benchmarkUtility.hpp>
```

The last include we have allows us to use command line command from the main. In particular we will use it to create a directory where we will save the output of the simulation.

```
#include <sys/stat.h>
```

All the ionic models headers are included in the benchmarkUtility.hpp file. We do not need to repeat them here.

At the beginning of the program we initialize MPI and the Epetra communicator. In this example the communicator will be explicitly used only to show some output on screen. The solver will use the communicator whenever needed and for the moment you do not need to worry about parallelism too much.

```
Int main ( Int argc, char** argv )
{
    //! Initializing Epetra communicator
    MPI_Init (&argc, &argv);
    boost::shared_ptr<Epetra_Comm>  Comm ( new Epetra_MpiComm (MPI_COMM_WORLD) );
    if ( Comm->MyPID() == 0 )
    {
        std::cout << "% using MPI" << endl;
    }
```

The `MyPID()` method returns the ID of the processor. Therefore only the master processor, with ID equal to zero is allowed to print out the message on screen.

Then the first thing we do is to create the folder where we save the output. We create a GetPot object that is capable of reading flags from the command line. In particular, we will launch the executable using for example the command `mpirun -n 2 Electrophysiology_test_benchmark -o OutputFolder`. The string "OutputFolder" after the flag -o will be read by the GetPot object and only the master processor will call the `mkdir` function in order to create a new folder on the hard drive. Make sure you have writing permissions in the folder of the executable! In case you don't set an output folder, by default a folder with the name "Output" will be created and used to save the solutions. Note that if you don't change the output folder, you are probably going to overwrite the solution of a previous simulation.

```
    GetPot commandLine ( argc, argv );
    std::string problemFolder = commandLine.follow ( "Output", 2, "-o", "--output" );
    // Create the problem folder
    if ( problemFolder.compare ("./") )
    {
        problemFolder += "/";

        if ( Comm->MyPID() == 0 )
        {
            mkdir ( problemFolder.c_str(), 0777 );
        }
    }
```

Next we define some typedefs. These are pretty much the same in all LifeV (but not everywhere!). It's good to start to have a look at them.

All type are defined with the suffix `_Type`. All shared pointer types are defined with the suffix `Ptr_Type`. Note that the monodomain solver is template on the mesh and on the ionic model.

Note also that I am using a typedef to the base class of the ionic model. Although the ElectroIonicModel is a pure abstract class, I will use specific constructors for the ionic models (see the benchmarkUtility.hpp file).

```
    // The mesh
    typedef RegionMesh<LinearTetra>                                    mesh_Type;

    //A distributed vector, LifeV wrapper of Trilinos Epetra_vector
    typedef VectorEpetra                                          vector_Type;
    typedef boost::shared_ptr<vector_Type>                        vectorPtr_Type;
```

```
    //The LifeV finite element matrix, wrapper of Trilinos Epetra_FECrsMatrix
    typedef MatrixEpetra<Real>                                          matrix_Type;
    typedef boost::shared_ptr<matrix_Type>                              matrixPtr_Type;

    // Boost function: takes as argumets
    // the time t,
    // the space coordinates (x,y,z)
    // the component of the vectorial function i
    typedef boost::function < Real (const Real& /*t*/,
                                    const Real &   x,
                                    const Real &   y,
                                    const Real& /*z*/,
                                    const ID&   /*i*/ ) >             function_Type;

        //Here we define the ionic model base type and its pointer
    typedef ElectroIonicModel                                           ionicModel_Type;
    typedef boost::shared_ptr<ionicModel_Type>                          ionicModelPtr_Type;

        //The monodomain solver is template on the mesh and the ionic model
    typedef ElectroETAMonodomainSolver< mesh_Type, ionicModel_Type >    monodomainSolver_Type;
    typedef boost::shared_ptr< monodomainSolver_Type >                  monodomainSolverPtr_Type;
```

As in the previous tutorial, we import the xml file with the parameters using Teuchos.

```
    Teuchos::ParameterList monodomainList =
             * ( Teuchos::getParametersFromXmlFile ( "MonodomainSolverParamList.xml" ) );
```

Note that in the xml file "MonodomainSolverParamList.xml" I have not specified the ionic model parameters. The first part of the list contains the monodomain parameters such as:

- surface to volume ration
- the longitudinal and transversal conductivity conefficients (improperly called diffusion here)
- The timestep, the final time and the savestep (that is the timstep for exporting the simulation)
- The order of the elements to be used in the simulation (P2 have not been test. Keep using P1.)
- The ionic model you wish to use. At the moment in the benchmark the following model are available:
  - AlievPanfilov
  - LuoRudyI
  - TenTusscher06
  - HodgkinHuxley
  - NoblePurkinje
  - MinimalModel
  - Fox (tested with timestep 0.0025 ms, RushLarsen method not implemented)
- The solution method to be chose between
  - splitting (first order operator splitting), here you can specify the number of subiteration for the reaction step
  - ICI
  - L-ICI
  - SVI
- Mass lumping

- The path to the directory of the mesh and the name of the mesh file.

The other parameters are used in the solver to set up the linear solver. As you can see, here we are using the Trilinos package AztecOO.

```
<ParameterList>
    <!-- Monodomain solver parameters -->
    <Parameter name="surfaceVolumeRatio"    type="double" value="1400.0"  /><!-- cm^-1 -->
    <Parameter name="longitudinalDiffusion" type="double" value="1.3342"  /><!-- kOhm^-1 cm^-1 -->
    <Parameter name="transversalDiffusion"  type="double" value ="0.17606"/><!-- kOhm^-1 cm^-1 -->
    <Parameter name="timeStep"              type="double" value="0.1"     /><!-- ms -->
    <Parameter name="endTime"               type="double" value="55."     /><!-- ms -->
    <Parameter name="saveStep"              type="double" value="1.0"     /><!-- ms -->
    <Parameter name="elementsOrder"         type="string" value="P1"      />
    <Parameter name="ionic_model"           type="string" value="TenTusscher06"/>
    <Parameter name="solutionMethod"        type="string" value="ICI"     />
    <Parameter name="subiter"               type="int"    value="1"       />
    <Parameter name="LumpedMass"            type="bool"   value="true"    />
    <Parameter name="mesh_name"             type="string" value="benchmark_05mm.mesh"/>
    <Parameter name="mesh_path"             type="string" value="./"      />

    <!-- LinearSolver parameters -->
    <Parameter name="Reuse Preconditioner"     type="bool"   value="true"/>
    <Parameter name="Max Iterations For Reuse" type="int"    value="80"/>
    <Parameter name="Quit On Failure"          type="bool"   value="false"/>
    <Parameter name="Silent"                   type="bool"   value="true"/>
    <Parameter name="Solver Type"              type="string" value="AztecOO"/>
    <Parameter name="OutputFile"               type="string" value="Solution"/>

        <!-- Operator specific parameters (AztecOO) -->
        <ParameterList name="Solver: Operator List">

                <!-- Trilinos parameters -->
                <ParameterList name="Trilinos: AztecOO List">
                <Parameter name="solver"   type="string" value="cg"/>
                <Parameter name="conv"     type="string" value="rhs"/>
                <Parameter name="scaling"  type="string" value="none"/>
                <Parameter name="output"   type="string" value="none"/>
                <Parameter name="tol"      type="double" value="1.e-10"/>
                <Parameter name="max_iter" type="int"    value="200"/>
                <Parameter name="kspace"   type="int"    value="100"/>
                <!-- az_aztec_defs.h -->
                <!-- #define AZ_classic 0 /* Does double classic */ -->
                <Parameter name="orthog"   type="int"    value="0"/>
                <!-- az_aztec_defs.h -->
                <!-- #define AZ_resid 0 -->
                <Parameter name="aux_vec"  type="int"    value="0"/>
        </ParameterList>

    </ParameterList>

</ParameterList>
```

Once the parameter list has been imported we can setup the ionic model. In particular we read from the imported list the ionic model we wish to use (if not found use the minimal model) that we pass to the `chooseIonicModel` function defined in the benchmarkUtility.hpp file. This function check the string defining the ionic model and create the correct object. As you can see this function returns a real value. This value is used to define the threshold at which we consider electrical activation. In particular, we wish to save the activation times on the mesh, that is I will

check the time at which the wave front is passing though each node. Each model may have a different activation value that we can set and it is set through the chooseIonicModel function.

```
    //read the ionic model from xml file
    std::string ionic_model ( monodomainList.get ("ionic_model", "minimalModel") );
    if ( Comm->MyPID() == 0 )
    {
        std::cout << "\nIonic_Model:" << ionic_model;
    }
        //create the desired ionic model
    ionicModelPtr_Type  model;
    Real activationThreshold = BenchmarkUtility::chooseIonicModel(model, ionic_model, *Comm );
```

Once we have the ionic model we can create the monodomain solver. The constructor take as arguments the name and the path of the mesh, a GetPot object and the ionic model. The GetPot is used to set up the preconditioner. In this case we do not have a datafile where we specify the preconditioner. If you would like to change the preconditioner you can create the datafile and pass it here in the monodomain constructor. By default we use ML.

```
        //read the mesh and meshpath from the xml file
    std::string meshName = monodomainList.get ("mesh_name", "");
    std::string meshPath = monodomainList.get ("mesh_path", "./");
        //create a GetPot Object (required to set up the preconditioner)
    GetPot dataFile  (argc, argv);
        //call the monodomain solver constructor
    monodomainSolverPtr_Type solver ( new monodomainSolver_Type ( meshName, meshPath, dataFile , model ) )
```

Now we need to setup the monodomain model. First we initialize all the variables in the model with the resting values as defined in the ionic model. Then we set up the parameters of the monodomain.

```
    //setup all the variable to M_restingConditions as defined in the ionic model
    solver -> setInitialConditions();
    //setup the parameters of the monodomain
    solver -> setParameters ( monodomainList );
```

We need to provide a fiber direction. In order to do this one has several options:

- define the fibers through a function
- give a constant value to the fiber direction
- load the fiber field from an external data (text file or HDF5 file)

In the benchmark the fiber are aligned in the z-direction therefore we can use the setupFibers method which takes as argument a `VectorSmall` (a LifeV vector type).

```
    VectorSmall<3> fibers;
    fibers[0] = 0.0;
    fibers[1] = 0.0;
    fibers[2] = 1.0;
    solver -> setupFibers ( fibers ); //generates the fiber field
```

The operators we are going to setup depends on the solution method we use. Therefore we first read from datafile the method.

```
std::string solutionMethod = monodomainList.get ("solutionMethod", "splitting");
```

and then we setup the matrices. I will assume now that we are going to use ICI. This is the most involved solution method since it requires two mass matrices, one full and one lumped. The solution methods are not embedded in the solver (I am thinking of writing some separate function to call this blocks of code each time).

First of all, let's see if you want to use a lumped matrix at all. It's recommended to use it and I assume you will use it!

The solver, at the moment, stores only one mass matrix. On the other hand the setParameters method, we called earlier, read if we wish to use a lumped mass matrix and set a boolean member. The `setupMassMatrix()` check if this members is true or false and setup the mass matrix accordingly. Therefore, if I want to use a lumped mass matrix and ICI , I create a pointer to another mass matrix (`hlmass`) where I will save the full mass matrix. I do:

1. setLumpedMassMatrix to false so that the solver will create a full mass matrix;
2. setup the full mass matri;
3. create a copy of the full mass matrix pointed by hlmass;
4. restore the setLumpeMassMatrix to true;
5. setup the lumped mass matrix
6. setup the stiffness matrix
7. setup the global matrix

This passages are done in the following way:

```
    bool lumpedMass = monodomainList.get ("LumpedMass", true);
    matrixPtr_Type hlmass;
    if( lumpedMass )
    {
        solver -> setLumpedMassMatrix(false);
                solver -> setupMassMatrix();
                hlmass.reset(new matrix_Type( *(solver -> massMatrixPtr() ) ) );
                solver -> setLumpedMassMatrix(lumpedMass);
    }
    solver -> setupMassMatrix(); //  M
    solver -> setupStiffnessMatrix (); //  K
    solver -> setupGlobalMatrix(); //  M/dt + K
```

Next we setup the HDF5 exporter to save the solution. The monodomain solver does not store the exporter. On the other hand, it can setup the exporter for you. More precisely, all the variables will be saved with the name "Variable"+componentNumber, that is: Variable0 (the potential), Variable1, Variable2 ... We need to pass to the setupExporter method, only the name of the file that will be created and in which folder we want to save. Then, we can use the modomain solver to export the solution.

```
ExporterHDF5< RegionMesh <LinearTetra> > exporter;
solver -> setupExporter ( exporter, monodomainList.get("OutputFile","Solution") , problemFolder);
solver -> exportSolution ( exporter, 0);
```

As we have just seen the monodomain provides a wrapper to save the solution on an HDF5 exporter. Recall that we also want to save the activation times. Therefore we need to create another exporter where we save the activation times. First I create a vector where I will store the activation times. As the times are nonnegative ($t_0 = 0$), I initialize the activation time vector to a negative value. The setting up the exporter requires to specify: 1) the processors ID (`setMeshProcId`) (not mandatory); 2) add the pointer of the variable we want to save (`addVariable`); 3) set the name of the file we are exporting (`setPrefix`); 4) set the folder in which we want to save the solution (`setPostDir`).

```
//initialize the activation time vector with negative values
vectorPtr_Type activationTimeVector ( new vector_Type ( solver -> potentialPtr() -> map() ) );
*activationTimeVector = -1.0;
//setup the exporter for the activation times
ExporterHDF5< RegionMesh <LinearTetra> > activationTimeExporter;
activationTimeExporter.setMeshProcId (solver -> localMeshPtr(), solver -> commPtr()->MyPID() );
activationTimeExporter.addVariable (ExporterData<mesh_Type>::ScalarField, "Activation Time",
            solver -> feSpacePtr(), activationTimeVector, UInt (0) );
```

```
activationTimeExporter.setPrefix ("ActivationTime");
activationTimeExporter.setPostDir (problemFolder);
```

We define the timestep and we read the savestep from the xml file. To actually save the solution at the correct times we check the iterations. We setup a counter for the time loop.

```
    Real dt(solver -> timeStep());
    Int iter = monodomainList.get ("saveStep", 1.0) / dt; //number of iterations between each save
    Int k (0); //time loop iteration counter
```

Before solving we need to specify the applied stimulus. In this example the applied stimulus is defined through an external function (defined in the benchmarkUtility.hpp file). To choose the correct function, which depends on the ionic model, I do

```
    function_Type stimulus; //declare function
    BenchmarkUtility::setStimulus(stimulus, ionic_model); //set up stimulus function
```

Now we can start solving. We do a time loop consisting of the following steps

1. update the applied current (some model may have the applied current in other equations!)
2. solve for the gating variables and ionic currents using Rush-Larsen if available
3. solve the potential equation using ICI (passing the full mass matrix)
4. update time
5. register the activation time
6. save the solution if needed

```
    for ( Real t = solver -> initialTime(); t < solver -> endTime(); )
    {
        //update the applied current vector
        solver -> setAppliedCurrentFromFunction ( stimulus, t );
        //solve for the gating variable if possible with Rush-Larsen
        if(ionic_model!="MinimalModel" && ionic_model!="AlievPanfilov" && ionic_model!="Fox")
            solver -> solveOneStepGatingVariablesRL();

        //else solve using forward Euler
        else
            solver -> solveOneStepGatingVariablesFE();

        //Solve the potential equation with ICI
        solver -> solveOneICIStep(*hlmass);

        //update loop counter and time
        k++;
        t = t + dt;

        //register activation time on the desired vector
        solver -> registerActivationTime (*activationTimeVector, t, activationThreshold);

        //export the solution only if we are at the right time
        if ( k % iter == 0 )
        {
            solver -> exportSolution (exporter, t);
        }
    }
```

Before concluding we still need to export the activation times and close the exporters.

```
    activationTimeExporter.postProcess (0); //export activation time
    activationTimeExporter.closeFile();     //close exporters
    exporter.closeFile();
```

Moreover if we would like to export also the fiber direction we can call the following monodomain solver method

```
    solver -> exportFiberDirection(problemFolder); //export fiber field in the output folder
```

If you have introduced a new ionic model and you wish to use it in the benchmark you need to modify only the benchmarkUtility.hpp file. In particular add the header of your ionic model

```
#include <lifev/electrophysiology/solver/IonicModels/MySuperCoolIonicModel.hpp>
```

Add in the the chooseIonicModel function the lines

```
    if ( ionic_model == "MySuperCoolIonicModel")
    {
        model.reset ( new MySuperCoolIonicModel() );
        activationThreshold = 17.39; //or whatever value you think is correct
        ionicModelSet = true;
    }
```

In principle, if you are using a biophysically detailed model you should not change the stimulus. On the other hand if you want you can add your own pacing protocol in the benchmarkUtilities.hpp file. If you would like to sue a specific, like S1-S2, consider using the ElectroStimulus class (see the test_pacing for an example of the usage of such class).

4.4 Creating the fiber field on a ventricular geometry

In this tutorial I will show you the details of the test_fibers. This test can be used to create a fiber field on arbitrary ventricular or biventricular geometries. Here we provide only an idealized left ventricular mesh. To use your own, you just need to set up correctly the boundary conditions and the left ventricle centerline. The details of the algorithm can be found in [18].

In this tutorial I will comment only the fundamental points. I believe you should be able to understand the code as there are many comments.

The algorithm requires to solve a Laplace equation. To set up the boundary conditions I use the `BCInteface` class. In this way I can define the boundary conditions on a datafile. The BCInterface class is template on a physical solver. To create a fiber field I don't want to rely on the monodomain solver. I inserted in the bc_interface module a DefaultPhysicalSolver that we can use here.

Then when I typedef the BCInterface I can do

```
    typedef BCHandler                                       bc_Type;
    typedef boost::shared_ptr< bc_Type >                    bcPtr_Type;

    typedef DefaultPhysicalSolver<VectorEpetra>             physicalSolver_Type;

    typedef BCInterface3D< bc_Type, physicalSolver_Type >   bcInterface_Type;
    typedef boost::shared_ptr< bcInterface_Type >           bcInterfacePtr_Type;
```

In this test we do not have an xml file. On the other hand I provided a datafile. We can read this datafile using GetPot. The datafile has a simple structure: it defines a section, called "problem", and several subsections: "boundary_conditions", "space_discretization", "solver" and "prec" (preconditioner). You can access to each of these sections as if they were folders and subfolders.

To load the datafile, you need to specify the name of the datafile after the flag -f (like for export folder). If the flag is not specified a datafile with the name data will be read. You can launch the program with the command `mpirun -n 2 Electrophysiology_test_fibers -f DataFileName -o OutputFolder`.

```
    GetPot command_line (argc, argv);
    const string data_file_name = command_line.follow ("data", 2, "-f", "--file");
```

```
    GetPot dataFile (data_file_name);
```

Since I'm copying the mesh in the data folder to the binary mesh through the makefile, for example, in the space discretization I can set

```
//datafile
   [../space_discretization]
    mesh_dir    = ./
    mesh_file   = idealized.mesh
```

I can read these two strings in the following way

```
    std::string meshName = dataFile( "problem/space_discretization/mesh_file", "" );
    std::string meshPath = dataFile( "problem/space_discretization/mesh_dir", "./" );
```

After reading the mesh name the mesh path we can load the mesh (remember to create a pointer first).

```
    meshPtr_Type meshPart (new mesh_Type ( Comm ) );
    MeshUtility::loadMesh (meshPart, meshName, meshPath); //from lifev/core/mesh/MeshLoadingUtility.hpp
```

The assembly of the stiffness matrix is carried out using `Expression Template Assembly`. You can refer to the tutorial in that module for further information.

Once the matrix has been assemble we can set up the boundary condition using the BCInterface. When filling the BCHandler we need to specify the section in which we find the subsection "boundary_conditions".

```
    bcInterfacePtr_Type BC ( new bcInterface_Type() );
    BC->createHandler();
    BC->fillHandler ( data_file_name, "problem" );
    BC->handler()->bcUpdate( *uFESpace->mesh(), uFESpace->feBd(), uFESpace->dof() );
```

Then we can apply the boundary conditions calling the BCManage.

```
    bcManage ( *systemMatrix, *rhs, *uSpace->mesh(), uSpace->dof(), *BC -> handler(), uFESpace->feBd(), 1.
```

Then boundary conditions in the datafile has been specified are follows. We need to provide a list where we set the name of each boundary. Then we create a subsection on the datafile with the name of the boundary in the list where we specify the boundary conditions. In particular we need to set the type (essential, natural, Robin, ...), the flag of the boundary, the mode (Full, Component, Normal, ...), the component on which the boundary condition is applied (numbering start from zero) (if Full component is the number of components in the problem), the function to assign on the boundary (may be a function of space and time). You can refer to the bc_interface module for further information. In this case we set up two simple Dirichlet boundary conditions on the endo and on the epicardium. On the base we set homogeneous Neumann conditions (we could remove this part since it does not affect the system).

```
    [./boundary_conditions]
    list = 'Endocardium Epicardium Base'

        [./Endocardium]
        type       = Essential
        flag       = 5
        mode       = Full
        component  = 1
            function  = '0.0'

        [../Epicardium]
        type       = Essential
        flag       = 10
        mode       = Full
        component  = 1
        function  = '1.0'
```

```
        [../Base]
        type        = Natural
        flag        = 40
        mode        = Full
        component   = 1
                function  = '0.0'

        [../]
```

To actually solve the system we need to create a LinearSolver and a preconditioner. The preconditioner can be set up using the datafile specifying in which section the parameters are found. In this case we use Ifpack.

```
    //setup the preconditioner
    typedef LifeV::Preconditioner             basePrec_Type;
    typedef boost::shared_ptr<basePrec_Type>  basePrecPtr_Type;
    typedef LifeV::PreconditionerIfpack       prec_Type; //If you wish you can chenge to PreconditionerML
    typedef boost::shared_ptr<prec_Type>      precPtr_Type;

    prec_Type* precRawPtr;
    basePrecPtr_Type precPtr;
    precRawPtr = new prec_Type;
    precRawPtr->setDataFromGetPot ( dataFile, "prec" );
    precPtr.reset ( precRawPtr );
```

On the other hand the linear solver requires a Teuchos parameter list, as in the previous tutorial. Now we do not have the xml file but all the parameters for the linear solver are defined in the datafile.

We could add the xml file to the folder. I decided to actually create the Teuchos::ParameterList from the datafile. This is achieved by calling the create List from GetPot. In that function I read the data from the datafile and I set those data in the solverList.

```
        //Filling the solver parameter list
    Teuchos::ParameterList solverList;
        createListFromGetPot(solverList, dataFile);
```

Once we have the Teuchos parameter list we can setup the linear solver

```
    //setup of the linear solver
    LinearSolver linearSolver;
        linearSolver.setCommunicator (Comm);
    linearSolver.setParameters ( solverList );
        linearSolver.setPreconditioner ( precPtr );
```

To solve the linear system we call the following

```
    linearSolver.setOperator (systemMatrix); //stiffness matrix with boundary conditions
    linearSolver.setRightHandSide (rhs);  //right hand sidde with boundary conditions
    linearSolver.solve (solution); //pointer to the vector where we want to store the solution
```

After we computed the solution of the Laplace equation, we define the sheet direction as the gradient of the solution. We use the gradient recovery procedure from Zienkiewicz and Zhu. We define three vector to store the components of the sheet direction and we call the ZZGradient function (for more details check the lifev/core/fem/GradientRecovery.hpp file).

```
vectorPtr_Type sx (new vector_Type ( uSpace -> map() ) ); //x component of the sheet field
vectorPtr_Type sy (new vector_Type ( uSpace -> map() ) ); //y component of the sheet field
vectorPtr_Type sz (new vector_Type ( uSpace -> map() ) ); //z component of the sheet field

*sx = GradientRecovery::ZZGradient (uSpace, *solution, 0); //fill the x component
*sy = GradientRecovery::ZZGradient (uSpace, *solution, 1); //fill the y component
*sz = GradientRecovery::ZZGradient (uSpace, *solution, 2); //fill the Z component
```

Next we want put the components in a vector with all the components. Therefore we can create the map for such a vector and initialize it. In the test I create a vectorial finite element space, since it will be needed anyway for the exporter. Then I can initialize the vector with the map of the vectorial space

```
vectorPtr_Type rbSheet ( new vector_Type ( vectorFESpace -> map() ) ); //rule-based sheet field
```

Now I want to fill this vector with the components computed using the gradient recovery. I can do this in parallel. First I compute the length of each vector on each processor using `MyLength()`.

Then looping on the local elements of the vector, I compute the global ID of the vector for each component and I assign it to the correct value computed with the gradient recovery. Eventually since the sheet field defines only the direction, I normalize the vector, using the `ElectrophysiologyUtility::normalize` function (see lifev/electrophysiology/util/ElectrophysiologyUtility.hpp). Note that the components in the `rbSheet` vector will be equally diveded among the processors. This means that if d is the number of elements on each processor corresponding to the vector `sx`, than the local vector `rbSheet` will have 3d elements store in the following way: first d elements for the x components, then d elements for the y component and finally `d` elements for the z component.

```
    int d = (*sx).epetraVector().MyLength(); //local number of element on each processor

    for ( int l (0); l < d; l++) //loop over the local elements
    {
        int i = (*rbSheet).blockMap().GID (l);        //Global ID of the x component
        int j = (*rbSheet).blockMap().GID (l + d);    //Global ID of the y component
        int k = (*rbSheet).blockMap().GID (l + 2 * d);//Global ID of the z component

        (*rbSheet) [i] = (*sx) [i];
        (*rbSheet) [j] = (*sy) [i];
        (*rbSheet) [k] = (*sz) [i];
    }
        //normalize the sheet field
    ElectrophysiologyUtility::normalize (*rbSheet);
```

In the remaing part of the test we follow the algorithm described in [18]. We read from datafile the left ventricle centerline, we build a local frame of reference and then we define the fiber using Rodrigues formula. The implementation uses the concept already detailed above. Therefore I conclude here the tutorial on the fibers.

Recall that on biventricular meshes you need to properly setup the boundary conditions. In particular you need to realize that you will have two different boundary conditions on the two side of the septum. For example you could set

```
    [./boundary_conditions]
    list = 'LVEndocardium Epicardium RVEndocardium RVSeptum'

        [./LVEndocardium]
        type       = Essential
        flag       = 5
        mode       = Full
        component  = 1
            function  = '0.0'

        [../RVEndocardium]
        type       = Essential
        flag       = 6
        mode       = Full
        component  = 1
            function  = '0.0'

        [../RVSeptum]
        type       = Essential
        flag       = 7
        mode       = Full
```

```
component  = 1
    function  = '1.0'

[../Epicardium]
type       = Essential
flag       = 10
mode       = Full
component  = 1
function  = '1.0'

[../]
```

Chapter 5

Testsuite

In this chapter I will describe briefly the testsuite in the module.

test_0DAlievPanfilovModel

In this test we solve the Aliev Panfilov model proposed in [1] with the parameters as in [12]

test_0DHodgkinHuxley

In this test we solve the classical Hodgkin-Huxley model as defined in [7].

test_0DFitzHughNagumoModel

In this test we solve the Fitzhugh-Nagumo model in the variants described in [4].

test_0DFoxModel

This model has been implemented to study alternans. Refer to [5] for more information.

test_0DGoldbeterModel

This is not an actual ionic model. The model describes the dynamics of Calcium oscillation. For more information refer to [6].

test_0DLuoRudyIModel

The classical milestone in electrophysiology. The Luo-Rudy phase I model is described in [9].

test_0DMinimalModel

The minimal model for human ventricular tissue has been proposed in [2].

test_0DMitchellSchaefferModel

A simple two current model [10].

test_0DNoblePurkinje

The first ionic model for Purkinje fibers as described in [7].

test_0DTenTusscher06Model

The second version of the human ventricular cell model. The implementation of the ionic model is a wrapper of the freely available C++ code Ten Tusscher has on her web page (http://www-binf.bio.uu.nl/khwjtuss/SourceCodes/) (Thank you a lot for this!). The model description can be found in [19].

test_benchmark

In this test we replicate the electrophysiology benchmark proposed in[13]. This test is capable of reproducing all the results in that paper. For more details see [14, 15].

The test consists in computing the activation times on a rectangular mesh with an external stimulus on an angle. Note that the provided mesh is rather coarse. Do not expect to be at convergence!

This test can use all the ionic models implemented in LifeV. The actual implemention on the other hand covers the following six models

- AlievPanfilov
- LuoRudyI
- TenTusscher06
- HodgkinHuxley
- NoblePurkinje
- MinimalModel
- Fox (tested with timestep 0.0025 ms, RushLarsen method not implemented)

Other models can be tested with small modification (see the tutorial). You can solve the monodomain using

- splitting (first order operator splitting), here you can specify the number of subiteration for the reaction step
- ICI
- L-ICI
- SVI

test_fibers

In this test we generate a rule-based fiber field, as well as a rule-based fiber field following the algorithm proposed in [18]. We provide a coarse idealized left ventricular geometry.

test_pacing

In this test we consider a small rectangular domain (one element thick) and we use the ElectroStimulus class to show how to use the pacing protocols. In particular we use the S1-S2-S3 stimulation protocol by default.

test_restart

In this test we show how to restart a simulation using the ElectroETAMonodomainSolver. In particular we run a simulation were we generated a spiral wave (given as data in the Solution.h5 file). In the test we load the previously computed solution before generating the spiral and we check that after generating the spiral we obtain the same solution.

test_ventricle

In this test we solve an excitation problem on the idealized left ventricular mesh with fiber direction created using the test_fibers. In particular we show how to import the saved fiber filed and how to initialize the solution on the whole endocardium. Note that this test uses the minimal model solved with SVI. On this coarse mesh the propagation will be superfast!

Bibliography

[1] Aliev, Rubin R., and Alexander V. Panfilov. "A simple two-variable model of cardiac excitation." Chaos, Solitons & Fractals 7.3 (1996): 293-301.

[2] Bueno-Orovio, Alfonso, Elizabeth M. Cherry, and Flavio H. Fenton. "Minimal model for human ventricular action potentials in tissue." Journal of theoretical biology 253.3 (2008): 544-560.

[3] Franzone, P. C., Pavarino, L. F., & Savaré, G. (2006). Computational electrocardiology: mathematical and numerical modeling. In Complex Systems in Biomedicine (pp. 187-241). Springer Milan.

[4] Franzone, P. C., Deuflhard, P., Erdmann, B., Lang, J., & Pavarino, L. F. (2006). Adaptivity in space and time for reaction-diffusion systems in electrocardiology. SIAM Journal on Scientific Computing, 28(3), 942-962.

[5] Fox, Jeffrey J., Jennifer L. McHarg, and Robert F. Gilmour Jr. "Ionic mechanism of electrical alternans." American Journal of Physiology-Heart and Circulatory Physiology 51.2 (2002): H516.

[6] Goldbeter, A., Dupont, G., & Berridge, M. J. (1990). Minimal model for signal-induced Ca2+ oscillations and for their frequency encoding through protein phosphorylation. Proceedings of the National Academy of Sciences, 87(4), 1461-1465.

[7] Keener, J. P., & Sneyd, J. (1998). Mathematical physiology (Vol. 8). Springer.

[8] Krishnamoorthi, S., Sarkar, M., & Klug, W. S. (2013). Numerical quadrature and operator splitting in finite element methods for cardiac electrophysiology. International Journal for Numerical Methods in Biomedical Engineering, 29(11), 1243-1266.

[9] Luo, Ching-hsing, and Yoram Rudy. "A model of the ventricular cardiac action potential. Depolarization, repolarization, and their interaction." Circulation research 68.6 (1991): 1501-1526.

[10] Mitchell, Colleen C., and David G. Schaeffer. "A two-current model for the dynamics of cardiac membrane." Bulletin of mathematical biology 65.5 (2003): 767-793.

[11] Munteanu, M., & Pavarino, L. F. (2009). Decoupled Schwarz algorithms for implicit discretizations of nonlinear monodomain and bidomain systems. Mathematical Models and Methods in Applied Sciences, 19(07), 1065-1097.

[12] Nash, Martyn P., and Alexander V. Panfilov. "Electromechanical model of excitable tissue to study reentrant cardiac arrhythmias." Progress in biophysics and molecular biology 85.2 (2004): 501-522.

[13] Niederer, Steven A., et al. "Verification of cardiac tissue electrophysiology simulators using an N-version benchmark." Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 369.1954 (2011): 4331-4351.

[14] Pathmanathan, P., Mirams, G. R., Southern, J., & Whiteley, J. P. (2011). The significant effect of the choice of ionic current integration method in cardiac electro-physiological simulations. International Journal for Numerical Methods in Biomedical Engineering, 27(11), 1751-1770.

[15] Pathmanathan, P., et al. "Computational modelling of cardiac electrophysiology: explanation of the variability of results from different numerical solvers." International Journal for Numerical Methods in Biomedical Engineering 28.8 (2012): 890-903.

[16] Perego, M., & Veneziani, A. (2009). An efficient generalization of the Rush-Larsen method for solving electrophysiology membrane equations. Electronic Transactions on Numerical Analysis, 35, 234-256.

[17] Quarteroni, A., Quarteroni, A. M., & Valli, A. (2008). Numerical approximation of partial differential equations (Vol. 23). Springer.

[18] S. Rossi, T. Lassila, R. Ruiz-Baier, A. Sequeira and A. Quarteroni. Cardiac electromechanics: a thermodynamically consistent model for ventricular wall thickening by orthotropic activation. *European Journal of Mechanics*. 2013. Accepted.

[19] Ten Tusscher, K. H. W. J., and A. V. Panfilov. "Alternans and spiral breakup in a human ventricular." Am J Physiol Heart Circ Physiol 291 (2006): H1088-H1100.