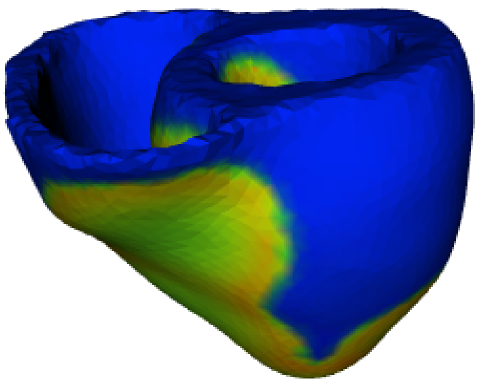
A solver for the Monodomain Heart Equations

Andreas V. Solbrå
a.v.solbra@fys.uio.no

August 2, 2013



Preface

During the summer of 2013 i have written an object oriented code which solves the partial differential equation (PDE) terms of the monodomain heart equations. The code solves the PDE by using the FEniCS software[1], and recieves as input the solution of the coupled system of ODEs in the heart equation, typically as provided by the packages goss[2] and gotran[3]. The purpose of this report is to detail the work done, as well as to note some of the pitfalls encountered underway.

If you are reading this, you are most likely my employer, or someone looking to use the code developed, in order to not have to re-invent the wheel. This report is mostly aimed at the latter. We start with a brief review of the theory (a more full description is given in [5]), and then spend some time going into details on the produced code. There will also be notes explaining the other minor files found among the produced code. Beware! This code was worked on for a mere six weeks, and never polished properly. There might be parts of the code that is unclear, ineffective or even completely wrong.

If you wish to make changes to or improve the code in some way, feel free to contact me, in order to become a collaborator on the code. If you only wish to use the code, it can be found at the gitub repository [4].

Contents

1 Introduction and theory **5**

1.1 Electrical activity in the heart . . . 5

1.1.1 Warm-up: equations for the torso . . . 5

1.1.2 Exitable tissue . . . 6

1.2 Bringing it all together . . . 7

2 Computational Details **8**

2.1 Operator splitting . . . 8

2.2 Solving the PDE part by finite elements . . . 8

3 External packages **9**

3.1 FEniCS . . . 9

3.2 gotran . . . 9

3.3 goss . . . 10

3.3.1 time solvers . . . 11

4 Description of the main program **12**

4.1 Goss_wrapper . . . 13

4.2 Time_solver . . . 13

5 Explanation of other files in the repo **14**

5.1 Implementations . . . 14

5.1.1 fe.py . . . 14

5.1.2 fem2D_fenics.py . . . 14

5.1.3 myocyte_activation.py . . . 14

5.1.4 The final run . . . 14

5.2 Other stuff . . . 15

5.2.1 extracellular_solver.py . . . 15

5.2.2 storing part of the solution vector . . . 15

5.2.3 purkinje . . . 16

5.2.4 nose test . . . 16

5.2.5 dolfin mesh from triangle . . . 16

5.2.6 grid and tetgen . . . 18

5.2.7 dolfin_animation_tools.py 19

1 Introduction and theory

This section will give a quick introduction to the problem we are looking to solve, and explain where the code produced will fit in a network of other packages.

1.1 Electrical activity in the heart

The heart beats as a result of electrical impulses produced by the heart cells themselves. This causes the intracellular electrical potential to increase, and this potential difference between the intracellular and the extracellular potential then spreads out like a diffusion process. It is useful to start off with some notation.

u_i	denotes the intracellular heart potential
u_e	denotes the extracellular heart potential
u_o	denotes the potential outside the heart (sometimes u_T is also used)
$v = u_i - u_e$	
H	denotes the heart domain
∂H	denotes the heart domain boundary (where the heart connects to the torso)
T	denotes the torso domain (outside the heart)
∂T	denotes the torso domain boundary (where the torso connects to the surroundings)

More will be introduced underway, but at the end these will be the most important ones. We will now go through a very brief derivation of the heart equations.

1.1.1 Warm-up: equations for the torso

We start with the spread of electrical potential in the torso. Assuming quasi-static conditions, we have from Maxwells equations that

$$\nabla \times E = 0, \tag{1}$$

which means that for some scalar potential u, we have

$$E = -\nabla u \tag{2}$$

The current is then given by the relation

$$J = ME \tag{3}$$

where M is the conductivity (this might be a tensor). This, of course, gives

$$J = -M\nabla u \tag{4}$$

If we assume that there is no sources or sinks for the potential inside the medium, and no build-up of charge, then for a small subvolume V with surface S, we have

$$\int_S n \cdot J \, dS = 0 \tag{5}$$

and so from the convergence theorem

$$-\int_V \nabla \cdot J \, dV = 0 \tag{6}$$

and since this is independent of the chosen domain, we have

$$\nabla \cdot J = 0, \tag{7}$$

and this means that

$$\nabla \cdot (M \nabla u) = 0 \tag{8}$$

What we have derived so far is just the standard heat equation. The torso should have the same potential on the boundary as the heart to ensure continuity of the potential, and if the torso is surrounded by air there should be no current flowing out of the body. The equations describing the torso is then

$$\nabla \cdot (M_T \nabla u) = 0 x \in T, \tag{9}$$
$$n \cdot M_T \nabla u_t = 0 x \in \partial T, \tag{10}$$
$$u_T = u_{\partial H} x \in \partial H. \tag{11}$$

1.1.2 Exitable tissue

The heart cells are called excitable, which means they can respond to an electrical stimulus. This ability enables an electric stimulation of one part of the heart to propagate through the muscle and activate the complete heart. This process happens through a depolarization: when the cells are at rest, there is a potential difference across the cell membrane. The stimulation causes the potential difference to go to zero or even further. This is a very fast process, and is followed by a slower repolarization that restores the difference. This is not handled by us at all, so I will skip the detailed on this, and just note that this creates a time dependent source term, and the equations describing this are

$$\nabla \cdot (M_i \nabla (u_e + v)) = \chi C_m \frac{\partial v}{\partial t} + \chi I_{ion} \tag{12}$$
$$\nabla \cdot (M_i \nabla v) + \nabla \cdot ((M_i + M_e) \nabla u) = 0 \tag{13}$$

where χ represents the area of cell surface per cell volume, and C_m is the capacitance of the cell membrane.

1.2 Bringing it all together

We have shown some of the relations that make out the heart equations. The complete system, in normalized units, is described by

$$\frac{\partial s}{\partial t} = f(s, v, t) \qquad x \in H \tag{14}$$

$$\nabla \cdot (M_i \nabla v) + \nabla \cdot (M_i \partial u_e) = \frac{\partial v}{\partial t} + I_{ion}(v, s) \qquad x \in H \tag{15}$$

$$\nabla \cdot (M_i \nabla v) + \nabla((M_i + M_e)\nabla u_e) = 0 \qquad x \in H \tag{16}$$

$$\nabla \cdot (M_o \nabla u_o) = 0 \qquad x \in T \tag{17}$$

$$u_e = u_o \qquad x \in \partial H \tag{18}$$

$$n \cdot (M_i \nabla v + (M_i + M_e)) = n \cdot (M_o \nabla u_o) \qquad x \in \partial H \tag{19}$$

$$n \cdot (M_i \partial v + M_i \nabla u_e) = 0 \qquad x \in \partial H \tag{20}$$

$$n \cdot M_o \nabla u_o = 0 \qquad x \in \partial T \tag{21}$$

This is called the **Bidomain model**. If we assume that the extra- and intracellular conductivity are linearly dependent, $M_e = \lambda M_i$, then the equations simplify greatly, and we are left with

$$\frac{\partial s}{\partial t} = f(s, v, t) \qquad x \in H \tag{22}$$

$$\frac{\lambda}{1+\lambda}\nabla(M_i \nabla v) = \frac{\partial v}{\partial t} + I_{ion}(v, s) \qquad x \in H \tag{23}$$

$$n \cdot (M_i \nabla v) = 0 \qquad x \in \partial H \tag{24}$$

This is called the **Monodomain model**, and this is what we will be looking to solve here. (22) and the source term in (23) is taken care of by the goss/gotran packages, and we will take care of the rest.

2 Computational Details

In this section we will go through some of more details on how to set up the problem in order to solve it.

2.1 Operator splitting

The meat of what we are looking to solve is the equation

$$\frac{\partial v}{\partial t} = \nabla \cdot (M_i \nabla) - I_{ion}(v, s) \tag{25}$$

we could in principle solve this in one round, but as we do not have complete information about $I_{ion}(v, s)$, we typically can only rely on goss to solve the equation

$$\frac{\partial v}{\partial t} = -I_{ion}(v, s) \tag{26}$$

for one time step. Fortunately, this is exactly what we need when we use the technique called operator splitting. The basic idea is the following: consider a problem of the form

$$\frac{\partial v}{\partial t} = (L_1 + L_2)v \tag{27}$$

$$v(0) = v_0 \tag{28}$$

We can then find a trial solution by first solving

$$\frac{dv}{dt} = L_1(u) \tag{29}$$

$$u(0) = v_0 \tag{30}$$

to Δt, then solve

$$\frac{dw}{dt} = L_2(w)w(0) = u(\Delta t) \tag{31}$$

and then use $w(\Delta t)$ as an approximation to $v(\Delta t)$. Our code uses a slighty more advanced method, and both methods are discussed thoroughly in [5], p. 71-82.

2.2 Solving the PDE part by finite elements

Is it a waste of time to write this? prolly.

3 External packages

Several packages is used in the code. Here is a brief overview of these.

3.1 FEniCS

FEniCS is a package for solving PDEs using the finite element method, and what our code uses to solve the PDE part of the heart equation. FEniCS has a relatively good documentation[1], so i will not go into detail here.

3.2 gotran

Gotran is a sympy-based packed for storing stimulation ODEs. ODEs in gotran format are typically stored in `.ode` files. From python, you will typically only call the function `load_ode(filename)`, where filename is the name of the .ode file you want to use. A nice feature of gotran is that is that we can explore the properties of the action potential through the terminal commands `gotranprobe` and `gotranrun`. Gotranprobe is used to check the properties of the ode. A typical run may look something like this

```
/simula_summer13$ gotranprobe myocyte.ode
Loaded ODE model ’myocyte’ with:
            Num states: 30
      Num field states: 1
        Num parameters: 2
  Num field parameters: 2
         Num variables: 2
Information of myocyte Gotran model
States: Ca_c, Ca_d, Ca_i, Ca_rel, Ca_up, F1, F2, K_c, K_i, Na_c, Na_i, O, O_C,
O_Calse, O_TC, O_TMgC, O_TMgMg, V, a_ur, d_L, f_L1, f_L2, h1, h2, i_ur, m, n,
pa, r, s
Parameters: g_K1, ist
```

which gives us information on the ode. This is a coupled system of 30 ODEs, with 2 parameters. The names of the states and parameters are usually very cryptic, as in this case. However, the potential difference is usally called V (sometimes a lowercase v). To all parameters are given default values, but they can be set as well. In order to check the action potential, we can use the `gotranrun` function.

This will look something like this

```
 /simula_summer13$ gotranrun myocyte.ode --plot_states V --code.language C
 --parameters ist -10 --tstop 300
```

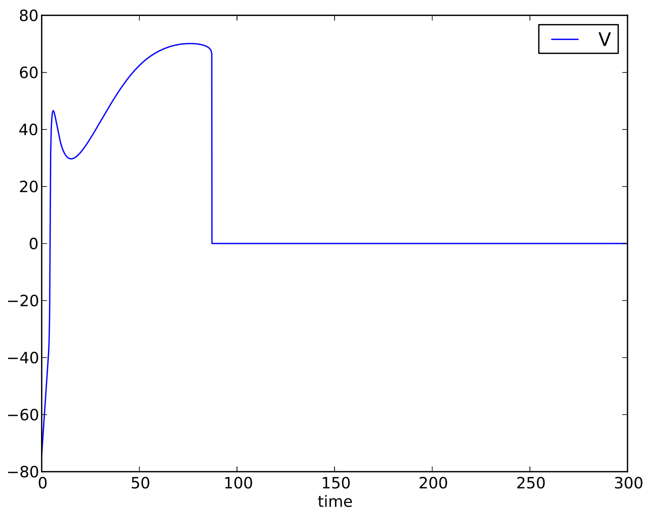


Figure 1: A possible activation potential.

which will produce a plot like the one shown in Fig. 1. The `plot_states` keyword selects what state to plot (typically we want this to be V or v), the parameter keyword sets the parameters. From the myocyte model, we need to apply some stimulation for the activation to take place. This varies from model to model. The tstop keyword select how long to solve for, and lastly, the code.language is set to C for improved speed.

3.3 goss

Goss is the package used to solve the ODEs stored by gotran. we initialize an ode by writing

```
from dolfin import *
from goss import *
from gotran import load_ode

#Fenics part
mesh = UnitSquareMesh(10, 10)
space = FunctionSpace(mesh, 'Lagrange', 1)
vertex_to_dof_map =  space.dofmap().vertex_to_dof_map(mesh)

### Setting up Goss/Gotran part
ode = jit(load_ode("myocyte.ode"))
N_thread = mesh.coordinates().shape[0]
fenics_ordered_coordinates = mesh.coordinates()[vertex_to_dof_map]
P0 = make_parameter_field(fenics_ordered_coordinates, ode)
solver = ImplicitEuler()
ode_solver = ODESystemSolver(int(N_thread), solver, ode)
```

Let's go through the goss/gotran part line by line. The first line loads the ode "myocyte.ode". The second line sets the number of states to solve for (goss actually solve these side by side). The third line produces an array with coordinates that is ordered in the same way as the solution vector produced by FEniCS and stores this to `fenics_ordered_coordinates`. The fourth line sets the parameters as illustrated in the previous section. Note that this is not a goss function, but a self-defined one. The fifth line sets the time solver method, and finally the last line creates the solver. The solver can then advance by `ode_solver.advance(dt,time)`, and the states can be set or get by `ode_solver.set_field_states(new_states)` and `ode_solver.get_field_states(state_vector)`. Note that `state_vector`must by preallocated, and will be filled with the solver states for the potential difference.

3.3.1 time solvers

goss has a variety of different time solvers. Here are some of them

- `ExplicitEuler()` - first order explicit euler. Very unstable, do not use.
- `ImplicitEuler()` - first order implicit euler. This one is very stable and good for testing
- `ThetaSolver()` - Theta rule. Has a parameter theta, that is initially set to 0.5, which gives a semi-implicit 2nd order scheme. Theta can also be manipulated directly. 1 gives Forward Euler, 0 gives backward Euler.
- `GRL2()` - 2nd order explicit method. This is quite stable. A (norwegian) description of the method can be found in [6] p. 53-54.

4 Description of the main program

In this section we will discuss the useage of the main program, `monodomain_solver.py`. We typically begin by creating a `Monodomain_solver`-object, and then we set all that should be set. A typical set-up will then look something like

```
    solver = Monodomain_solver(dt=0.1)
    solver.set_geometry(geometry)
    solver.set_source_term(goss_wrapper)
    solver.set_M(tensor_field)
    solver.set_boundary_conditions()
    solver.set_time_solver_method(Time_solver('CN'))
    solver.set_initial_condition(V_FEniCS_ordered)
```

let us go through these statements.

1. The first line creates the solver. The time step should be set here.
2. The second line sets the geometry for the problem. The input can be either a string, which will be interpreted as a path to a .xml mesh file, a Mesh object or a list of up to three integers. The last case creates a unit line/square/cube mesh with as many elements in each direction as input.
3. the thrid line sets the source term. The input can be either a Goss_wrapper object (see below) or a function. The function is simply interpreted as the right hand side of the equation
$$\frac{dv}{dt} = f(v) \tag{32}$$
4. the fourth line sets the conductivity tensor. The input should be a $n \times n$ tuple where n is the dimension of the geometry.
5. the fifth line should set the boundary conditions, but really this does nothing at this point, and the boundary is just assumed to adhere to van Neumann conditions.
6. the sixth line sets the time solver method. It takes as input a `Time_solver` object (see below).

Once the system is initialized, we can solve by calling

```
solver.solve(T, savenumpy=save, plot_realtime=True)
```

where Tis the time we want to solve to. savenumpy set whether or not to store the results, plot_realtime sets whether or not to plot at runtime using the viper tool.

The default program in monodomain_solver.py runs the solver with a self-defined source term ($f(v) = -v$).

4.1 Goss_wrapper

the goss wrapper is meant to be a sort of contract to ensure that there is some function to advance the solution (like an interface in java). The advance function is defined by the user, but should typically include the goss method forward(dt,time), which advances the solution in time from time to time+dt. Examples of advance functions can be found in fe.py and fem2D_fenics.py.

4.2 Time_solver

Time solver typically takes as input 'FE', 'BE' or 'CN'. Crank-Nicolson is second order, backward euler is the most stable, dont use forward euler for our problems.

5 Explanation of other files in the repo

The repo contains a lot of files aside from the monodomain solver, the extracellular solver and the torso solver. In this section I will explain some of them, and detail the work done. They can be split into to groups; some are implementations of the solvers for various ODEs and geometries, others are not.

5.1 Implementations

Several of the programs use the monodomain solver. Here are some of the notable ones.

5.1.1 fe.py

This was the first implementation using the goss package. Notably, unless i have since fixed the code, the advance function handles the ordering difference between the mesh.coordinates()-vector and the solution vector. This is very inefficient, and it's better to fix the orderering when calling `set_parameter_field(...)`. The program uses the myocyte model, and add a stimulus on the cells in a certain area. The stimulus then spreads as diffusion causes nearby cells to activate.

5.1.2 fem2D_fenics.py

This code uses a slightly more advanced model for some of the cells in the middle of the grid, difransesco.ode, which causes the cells to spontaneously activate. The activation then spreads through the other cells in the same way as fe.py

5.1.3 myocyte_activation.py

this program selects some areas as nodes in a purkinje network (see below), and adds a stimulus on these cells. The stimulus then spreads through the rest of the heart. This simulation uses a 3D heart mesh. The front page picture is from this simulation.

5.1.4 The final run

In the folder "the_final_run" there is the python file `myocyte_activation_redoux.py`, which solves the monodomain heart equations, as well as the elliptical equation for the rest of the torso;

$$\nabla \cdot (M_i \nabla v) + \nabla \cdot (M \nabla u_e) = 0 \tag{33}$$

where $M = M_e + M_i$ inside the heart, and $M = M_o$ outside the heart. The tensors in this case are taken from a simulation using the fiberrules package (ask around at simula to get access to this). In order to set M, some really roundabout functions are used, `get_inner_heart_tensor()` and `get_torso_tensor()`. What they do is to extract the 9 different scalar fields found in the 3 vector fields, and manually compute the cross product of the vectors. I'm not sure why this was neccessary, but not doing it in this way would cause the program to crash, saying that the function was not defined on the domain. This program marks the end of my intership.

5.2 Other stuff

Here we'll explain some of the other code.

5.2.1 extracellular_solver.py

This program solves the elliptical equation for the torso;

$$\nabla \cdot (M_i \nabla v) + \nabla \cdot (M \nabla u_e) = 0 \tag{34}$$

where $M = M_e + M_i$ inside the heart, and $M = M_o$ outside the heart. Essentially this code only does the solution step, and the user must take care of the map from the torso to the heart for v, as well as setting up the tensors for the different domains. For an example, see "the final run".

5.2.2 storing part of the solution vector

In some cases, we might not be able to store the entire solution vector from a simulation. This might be simply because the solution takes up too much space. There are several ways of doing this, and I will only propose one here. We begin by storing the coordinates for the point we would like to save

```
points = np.array([[0.0,0.0,0.0], [1.0, 1.0, 1.0]])
```

We then need to find the appropriate vertices in the mesh. In 2D we can use the FEniCS function `closest_point()`, but this does not work in 3D. Rather, we can do it like this

```
n = points.shape[0]
for i in range(n):
    point = points[i]
    distance_vec = mesh.coordinates() - point
    distances = distance_vec[0]**2 + distance_vec[1]**2 + distance_vec[2]**2
    point_idx = np.argmin(distances)
    dof_idx = vertex_to_dof_map[point_idx]
    point_to_store[i] = dof_idx
```

now that we have these points. We can at each time iteration store them by

```
u_array = solver.v_n.vector().array()
u_array_save = u_array[points_to_store]
np.save(filename,u_array_save)
```

this will save the points in binary form, to a file with the name given in `filename`.

5.2.3 purkinje

The spontaneous heart activation starts in the atrium, then passes through a common point (the Atrioventricular node) before its spreads to the ventricles. The spread happens mainly through the so-called purkinje fibers (the propagation is several hundred times faster through these cells than regular heart tissue). The purkinje files models such nerves based on the same heart mesh used in myocyte_activation. The leaf nodes found by this model are used for the activation is this program.

5.2.4 nose test

Nose tests is a simple tools for performin unit testing of the code. One can test such things as error convergence and constant solution tests. To run the tests, one can type

```
nosetests -a '!skip'
```

the extra commands are because the testing tool picks up some fenics functions that we do not want to test.

These are somewhat outdated at this point, and should be overhauled, but I'm not sure if there will be time for this during my stint.

5.2.5 dolfin mesh from triangle

Several tools exists to make and manipulate meshes. One such tool is triangle, which is available from apt-get. Triangle can take .poly files, which contain meshes and mesh functions, and convert these to other files. Here is a description on how to use it with dolfin. We start by entering into the terminal

```
triangle -a0.01 -A -q28 -p atrium2D.poly
```

where in this case the mesh file is called atrium2D.poly. This creates the files atrium2D.1.edge and atrium2D.1.node files. we can then create a dolfin mesh by entering either of the following commands

```
dolfin-convert atrium2D.1.node atrium2D.xml
dolfin-convert atrium2D.1.ele atrium2D.xml
```

This will create the files atrium2D.xml and atrium2D.attr0.xml, where the former is a dolfin mesh and the latter is a mesh function. In order to visualize this we can use the following code.

```
 from dolfin import *
mesh = Mesh('atrium2D.xml')
values = MeshFunction('double', mesh, 'atrium2D.attr0.xml')

#plot(mesh)
plot(values)
interactive()
```

Mesh functions are defined on the node values. If we want to for instance use the function to define the conductivity tensor, we need a function that is defined on the vertexes. We can use the following hack to translate the values to a vertex values function

```
 p = Function(V)

### hack to assign the meshfunction to vertex values:
dim = 2
data = values
values = values.array()
mesh.init(dim)
vertices = type(data)(mesh, 0)
vertex_values = vertices.array()
vertex_values[:] = 0
con20 = mesh.topology()(dim,0)

for facet in xrange(mesh.num_faces()):
  if values[facet]:
    vertex_values[con20(facet)] = values[facet]

vertex_to_dof_map = V.dofmap().vertex_to_dof_map(V.mesh())
new_vertex_values = np.zeros(len(vertex_values))
new_vertex_values = vertex_values[vertex_to_dof_map]

# Put any function of the point values here:
new_vertex_values = 1.*(new_vertex_values!=3)

p.vector().set_local(new_vertex_values)

# visualize the vertex function
plot(p)
interactive()
```

This code is also included in the file `testcode.py`.

5.2.6 grid and tetgen

The folder `/grid/meshes` contains, among other things the file reference.poly which is similar to the triangle file, but 3-dimensional. To convert this to a dolfin mesh, we can use the following method.

dolfin has a standard converter to xml files, called by

```
dolfin-convert <infile> <outfile>
```

Normally we create the .ele and .node files by calling

```
tetgen -qA -a0.01 -P reference.poly
```

The problem is that dolfin interprets, the .ele and .node files as 2d triangle meshes, and projects them onto the x,y-plane. We can circumvent this by also outputing a .mesh file from tetgen, by inputing

```
tetgen -qA -a0.01 -g reference.poly
```

(i.e. add -g), which will create the file "reference.1.mesh". This can be converted to a 3d mesh by calling

```
dolfin-convert reference.1.mesh reference.xml
```

There is also a mesh function in the .poly-file. To obtain this, we can use the following hack:

1) run

```
dolfin-convert reference.1.ele func.xml
```

2) The function will now be stored in func.attr0.xml. However, it will have the wrong dimension. The solution is simply to open the file in any text editor, change 'dim="2"' to 'dim="3"' on the 4th line, and save the changes.

3) the function can now be plotted normally using standard commands such as

```
from dolfin import *
mesh = Mesh('reference.xml')
meshfunc = MeshFunction('double', mesh, 'func.attr0.xml')
plot(meshfunc)
interactive()
```

5.2.7 dolfin_animation_tools.py

This file contains some useful functions for visualizing the results in mayavi, and saving the results. Note that this only works for unit squares and unit cubes. General meshes are not supported.

References

[1] *FEniCS home page*. July 2013. URL: www.fenicsproject.org.

[2] *goss repository*. July 2013. URL: https://bitbucket.org/johanhake/goss.

[3] *gotran repository*. July 2013. URL: https://bitbucket.org/johanhake/gotran.

[4] *source code repository*. July 2013. URL: https://github.com/andreavs/simula_summer13.

[5] Joakim Sundnes et al. *Computing the Electrical Activity in the Heart*. Ed. by T. J. Barth et al. Springer, 2006. ISBN: 978-3-642-07005-1.

[6] Else Bergene Zakariassen. “sammenligning av numeriske metoder for monodomene- og bidomenemodellen”. MA thesis. University of Oslo.