

2.1 Introduction

Mathematical modeling in cardiology is a challenge not only at the level of model development, but also at the level of its implementation. Models complexity, abundance of numerical methods, high prerequisite for programming skills - all this often requires researchers to look for ready-made solution that gives the opportunity to mainly focus on the scientific and not on programming questions. Fortunately, over the past several years, several high-quality computing packages have been proposed with the aim to help in this matter [1, 2]. They allow to cover a wide range of scientific problems and relieve the end user of a significant portion of programming work. However, a significant part of these packages implement a large number of complex mathematical methods, including finite element methods (FEM) to create high-precision modeling tools that implement state-of-the-art approaches in the domain. This poses a number of problems for the user, such as the search for powerful computing resources and the generation of high-quality computational meshes. In addition, such heavyweight and feature-rich packages a high entry barrier and require a long time for learning how to use them. The developer of the myokit package [3] notes these problems and offers myokit as a simpler interface for solving a wide range of electrophysiological problems.

The package described in this chapter is not aimed to become multi-functional instrument for a various tasks in cardiac modeling and we deliberately focused our efforts on simplified modeling approaches that test a wide variety of hypotheses. We called it Finitewave and its main idea is to provide user with easy-to-use software to run large-scale tissue or even organ-level electrophysiological simulations with a few lines of Python code. Another notable feature - using of weight matrix in the finite difference scheme which creates a flexible way to model tissue with a wide variety of properties including fibrosis.

2.2 Basic elements

Only two classes are required to run the basic Finitewave script: the `CardiacTissue` class which creates a medium object and the `CardiacModel` class which is represented as one of the electrophysiological models available to the user. Both classes combine the logic of the package and provide access to other important functions related to the subject area. Here we provide a brief description of the most significant elements of these classes, thus revealing the concept of how the package works.

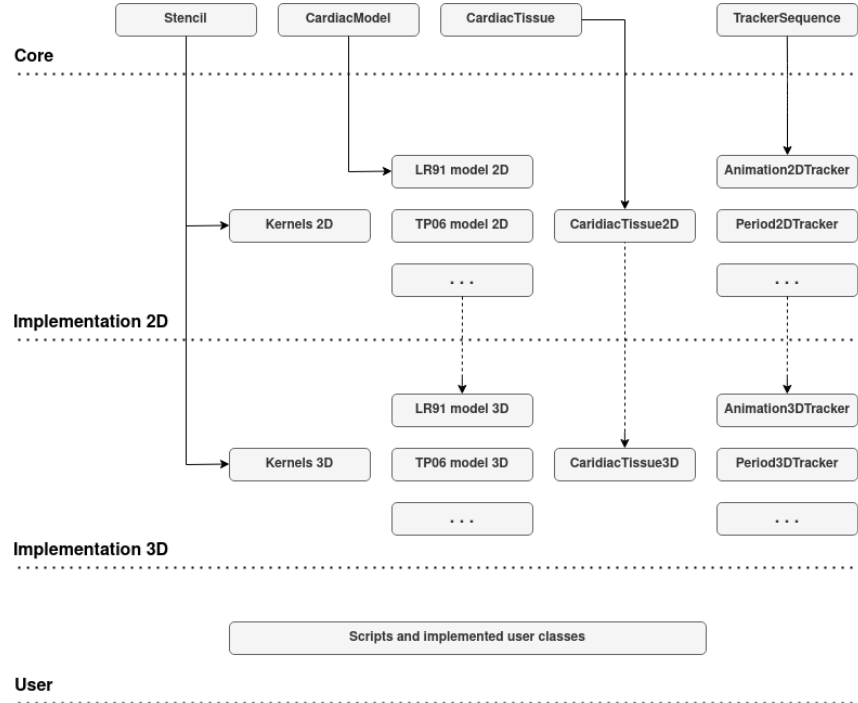


Figure 2.1: A simplified package structure that includes four levels: the core level includes base and abstract classes; implementation level is subdivided into 2D and 3D versions, containing a ready-made set of models, trackers and tools to get started quickly. The last user level means any programs written by the user, including implemented classes built into the general package hierarchy.

2.2.1 CardiacTissue class

Finitewave works with 2D or 3D medium and represents it as finite-difference regular mesh. This approach can be quite rough in approximating the boundaries of the target object, but is quite simple to generate using standard programming instruments. In the framework, the interaction with the medium is carried out through the CardiacTissue class, which includes three components: mesh, conductivity and fibers. An important component of the class is the weights generation method that together with finite-difference stencils determines the final form of the parabolic equation.

2.2.1.1 Attribute: Mesh

Mesh ('mesh' attribute) is a 2D or 3D integer array, defining the geometry of the medium itself (Fig. 2.2). Three different values represents the possible states of the

single node in the mesh: cardiomyocyte (0), boundary (1) and fibrosis (2). In terms of the numerical scheme boundary fibrosis and processed in the same way, but their separation is reasonable for visualization convenience. The only requirement for mesh generation is the placement of at least one layer of boundary nodes along the edges, which is easy to fulfill by calling the `add_boundary()` method.

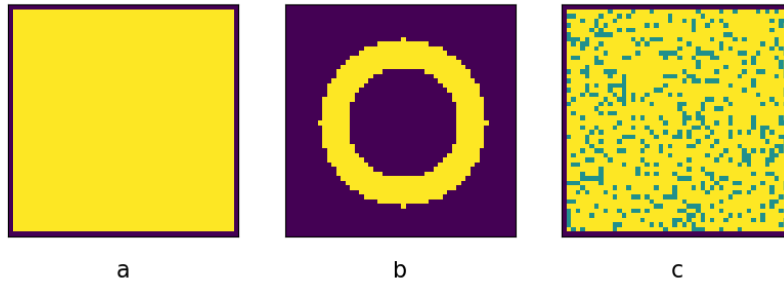


Figure 2.2: Arrays representing 50x50 nodes meshes suitable for usage in Finitewave: pure mesh (yellow) with the boundary (a), ring-like mesh with the complex boundary (b), mesh with the diffuse fibrosis (cyan) marked separately from the boundary nodes (c).

2.2.1.2 Attribute: Conductivity

Another way to model propagation delay is to create low-conductivity areas by reducing the diffusion coefficient in specific place (Fig. 2.3). Conductivity ('conductivity' attribute) is a 2D or 3D float array with the values in range from 0 (no conductivity) to 1 (full conductivity), this value is multiplied with the diffusion coefficient of the model to get an actual diffusion value for the region. Despite the fact that this approach has several disadvantages compared to non-conducting obstacles, it is a simple way to simulate fibrosis or can be used in optimization problems for characterizing the conductivity of the medium [4].

2.2.1.3 Attribute: Fibers

Myocardial tissue contains a complex fiber architecture characterizing the anisotropy of the medium. Together with the diffusion coefficients of the model, this has a significant effect on the speed of wave propagation in the medium (Fig. 2.4). Finitewave represents fibers field ('fibers' attribute) as 3D or 4D float array which specify 2D or 3D vector of the fiber direction for every node in the mesh.

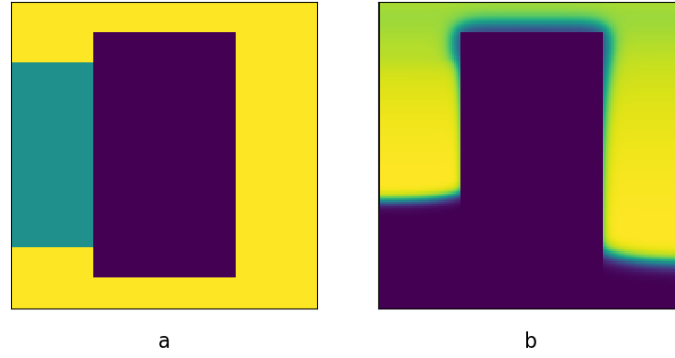


Figure 2.3: Conductivity array with the low-conductivity area on the left side (a) and potential map obtained from the calculations with the upper border stimulation which represents the specific time frame (b). Full-conductivity (conductivity = 1) and low-conductivity (conductivity = 0.5) areas are separated by the obstacle to isolate the excitation wave fronts.

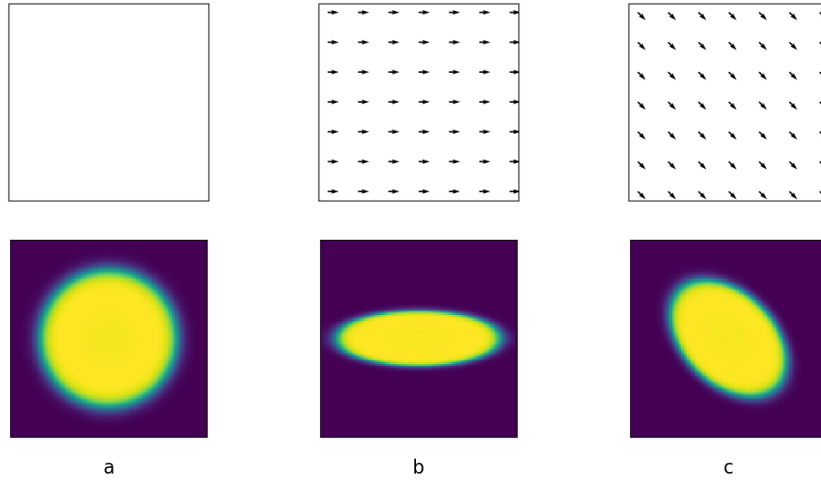


Figure 2.4: Anisotropic propagation with the central point-like stimulation: isotropic propagation with no fibers (a), fibers are elongated horizontally (b), fibers are elongated at an angle of 45° .

2.2.2 CardiacModel class

Cardiac model - is a central class that combines all the necessary components for making calculations. Its crucial point is the model equations implementation, as well as the main computational loop, which sequentially calls up the necessary

functionality (stimulation, trackers, commands). The main components of all the CardiacModel-inherited are kernels (numerical schemes implementation), StimSequence (stimulation protocol), TrackerSequence (runtime data gathering) and CommandSequence (runtime interactions with the model).

2.2.2.1 Kernels

All the model implementations are encapsulated in separate computational kernels that are called by the main computational loop. There are two types of kernel used in the model implementation: diffuse component kernels which describe the propagation process between the nodes (cells) and model variables kernels which specify the equations of the specific model. The main reason for the encapsulation of kernels is to solve the problem of speeding up the calculations. In this case, the entire model numerical implementation is a monolithic code that is taken out of the Python CardiacModel class and can be executed both using third-party libraries and on a separate device such as graphics cards.

2.2.2.2 StimSequence

Electrical stimulation of the cardiac tissue is the main mechanism that triggers the excitation wave. In Finitewave it is set by a sequence of stimuli that exist in two variants: voltage stimulation (StimVoltage class hierarchy), which sets the initial voltage value at given nodes, and current stimulation (StimCurrent class hierarchy). In the case of stimulation by current, there is a gradual accumulation of voltage in the given nodes, the duration of which is set by the user. Stimulations can go one after another, being triggered at a time set by the user. For this purpose, the StimSequence class is used, which receives a list of stimulations and executes them in the main computational loop.

2.2.2.3 TrackerSequence

Usually, it is necessary to know not only the final result of the calculation, but also the dynamics of the processes. Finitewave implements a large number of different tools that allow to solve the problem of collecting information during the calculation. Examples include: recording the dynamics of changes in model variables (MultiVariableTracker), activation maps recording - the time of the first arrival of the excitation wave at the mesh nodes (ActivationTimeTracker), recording the impulse repetition period at the set nodes (PeriodTracker), which is usually used to measure the period of spiral waves. Some trackers allow to create "snapshots" of the model state for later work with them, for example, AnimationTracker records frames with a specified time step and allows to create animations of the calculation progress.

If necessary, the user can implement a custom Tracker class based on the framework architecture and easily embed it into the TrackerSequence for execution.

2.2.2.4 CommandSequence

It is hard to take into account all possible actions user may want to perform during the calculation. Also some of them may disrupt the standard algorithm of work and require a special approach. Partially this can be done by with the CommandSequence class, which has an access to all the model class attributes and methods. As an example, user may want to stop calculations when the wave front reaches the specific mesh position even if the calculation time has not ended. Another possible example - user wants to gradually increase the size of the obstacle (obstacles) in the medium which requires recalculation of the weights for the numerical scheme. Those tasks as well as many other can be done using Command and CommandSequence classes - a flexible way to influence the calculation behavior.

2.3 Weighted numerical scheme

Finitewave is based on the finite difference method, which makes possible to significantly simplify and speed up computational work and flexibly formulate various situations associated with the modeling electrophysiological processes in cardiology. This approach has a number of significant drawbacks, but is best suited for fast hypothesis testing, generating large amounts of data, and learning the modeling basics. Taking into account the large number of possible conditions (complex boundaries, fibrosis, the presence of anisotropy of the medium), its technical implementation is still a challenge for programmers. Here we present our own weights-based approach to solve this problem.

2.3.1 Basic formulation

Monodomain representation of the wave propagation equation can be formulated as follows:

$$\frac{\partial u}{\partial t} = \nabla(D\nabla u) - I, \quad (2.1)$$

$$\vec{n}\nabla u = 0, \quad (2.2)$$

where u – is the transmembrane voltage, D – is the diffusion matrix for anisotropic tissue and constant for isotropic tissue, I – is the sum of all ionic currents used in the model which also contains at least one or more equations describing the changes in the remaining variables of the models associated with the behavior of

ion channels, but we omit its consideration, since they are not part of the weighting scheme and are solved explicitly by the Euler's method.

To solve this equation we must replace continuous derivatives with finite differences with the chosen time and spatial discretization parameters (steps). The solution u will be found for each node of the selected regular mesh. For convenience and brevity, we will consider only the two-dimensional formulation of the isotropic and anisotropic cases.

2.3.1.1 Isotropy

Isotropy of the medium means the same speed of the excitation wave propagation regardless of its direction. Although the cardiac tissue itself is anisotropic at the macroscopic level, this condition makes it possible to simplify the consideration of some problems, shifting the emphasis on other possible conduction effects. Consideration of an isotropic medium means the absence of a diffusion matrix in 2.1 and the presence of $D = \text{const}$ instead. Thus, the expression $\nabla(D\nabla u)$ can be reduced to $D\Delta u$.

We replace the partial derivatives with finite differences with the time and spatial steps τ and h :

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\tau} = D \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{h^2} + D \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{h^2} - I_{i,j}^n \quad (2.3)$$

The equation 2.3 is solved at each time step ($n, n+1, n+2, \dots$) for each grid node and its specific form significantly depends on the presence or absence of boundaries around the considered potential $u_{i,j}$.

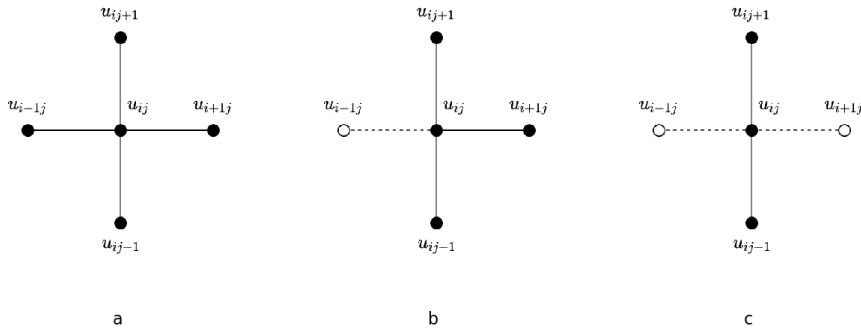


Figure 2.5: 5-points laplace: all neighbors exist (a), left node is excluded (b), left and right nodes are excluded (c).

Taking into account the presence of boundaries and non-conductive obstacles, we can consider several possible cases for the numerical scheme (Fig. 2.5). Case (a) corresponds to equation 2.3 and means the presence of all neighbors of the node $u_{i,j}$.

In case (b), the left node is excluded and we cannot use the value of the trans-membrane potential from it. Here we take the missing value from the boundary conditions:

$$\frac{u_{i+1,j}^n - u_{i-1,j}^n}{2h} = 0 \quad (2.4)$$

$$u_{i,j}^{n+1} = D\tau \frac{2u_{i+1,j}^n - 2u_{i,j}^n}{h^2} + D\tau \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{h^2} - u_{i,j}^n - I_{i,j}^n \quad (2.5)$$

In the case (c), there are no left and right nodes, which reduces the two-dimensional problem to the one-dimensional one and we completely exclude the i-term and from the equation:

$$u_{i,j}^{n+1} = D\tau \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{h^2} + u_{i,j}^n - I_{i,j}^n \quad (2.6)$$

Other cases that correspond to the j-term can be treated in the same way.

2.3.1.2 Anisotropy

Anisotropy is an important property of cardiac tissue and means a different propagation speed of the excitation wave in the medium, which will depend on its direction. Physiologically, this behavior is associated with the elongated shape of cardiomyocytes and the presence of a different number of gap junctions along and across the cell connections. We can imagine cardiac tissue in the form of rotating fibers, along which the wave propagation speed is higher.

In anisotropic case, we can rewrite the diffusion term as ∇q , where the flux is $q = D\nabla u$. We write central difference for ∇q using 4 points halfway between grid points (Fig. 2.6 a)

$$\nabla q \approx \frac{q_{i+\frac{1}{2},j} - q_{i-\frac{1}{2},j}}{h} + \frac{q_{i,j+\frac{1}{2}} - q_{i,j-\frac{1}{2}}}{h}. \quad (2.7)$$

where, the approximation for fluxes at halfway points are given by

$$\begin{aligned} q_{i+\frac{1}{2},j} &= d_{1,i+\frac{1}{2},j} \frac{u_{i+1,j} - u_{i,j}}{h} + \\ &+ d_{2,i+\frac{1}{2},j} \frac{0.5(u_{i,j+1} + u_{i+1,j+1}) - 0.5(u_{i,j-1} + u_{i+1,j-1})}{2h}, \\ q_{i,j+\frac{1}{2}} &= d_{3,i,j+\frac{1}{2}} \frac{u_{i,j+1} - u_{i,j}}{h} + \\ &+ d_{2,i,j+\frac{1}{2}} \frac{0.5(u_{i+1,j} + u_{i+1,j+1}) - 0.5(u_{i-1,j} + u_{i-1,j+1})}{2h} \end{aligned} \quad (2.8)$$

and the same approximations at $(i - \frac{1}{2}, j)$ and $(i, j - \frac{1}{2})$. To use the scheme we define the diffusion matrix at halfway points by approximation from neighbour points or initially generate approximated diffusion at these points. These approximation scheme labeled as asymmetric because of the different treatment of the x - and y -differential in each point.

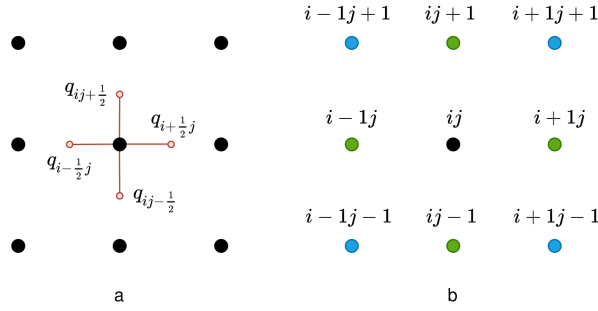


Figure 2.6: Asymmetric scheme for anisotropic diffusion: fluxes are defined on halfway points (a); there are different treatment for missing direct (green) and diagonal (blue) neighbours (b).

If some of neighbouring points are missing (via boundaries or fibrosis) we use the following rules (Fig. 2.6 b):

- for direct neighbours $(i + 1, j)$, $(i - 1, j)$, $(i, j + 1)$ or $(i, j - 1)$, the flow from that direction is set equal to 0;
- for diagonal neighbours $(i + 1, j + 1)$, $(i + 1, j - 1)$, $(i - 1, j + 1)$ or $(i - 1, j - 1)$, corresponding x - and y -differentials are approximated using existing points. For example, if point $(i + 1, j + 1)$ is missing, y -differential at $(i + \frac{1}{2}, j)$ is approximated as $\frac{u_{i,j+1} - u_{i,j-1}}{2h}$ and x -differential at $(i, j + \frac{1}{2})$ is approximated as $\frac{u_{i+1,j} - u_{i-1,j}}{2h}$.

2.3.1.3 Heterogeneity

Even in the absence of fibers, the excitation wave propagation speed can differ in different areas of the medium. In this case, we are dealing with a heterogeneous medium with the variable diffusion coefficient. In the isotropic case, the mathematical formulation of equation 2.1 will have additional terms. It should be noted that this is not the only approach to describing a heterogeneous medium [5, 6].

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} \frac{\partial D}{\partial x} + \frac{\partial u}{\partial y} \frac{\partial D}{\partial y} + D\Delta u - I, \quad (2.9)$$

The above scheme for anisotropy can take into account the heterogeneity of the medium together with anisotropy due to the approximation of the diffusion coefficients.

2.3.2 Weighted scheme

Implementation of all possible patterns presented above is possible without programming each individual case separately. To do this, we can represent the formulation 2.1 as a linear combination:

$$\begin{aligned}
 u_{i,j}^{n+1} = & w_1 u_{i+1,j}^n + w_2 u_{i,j}^n + w_3 u_{i-1,j}^n + w_4 u_{i,j+1}^n + w_5 u_{i,j}^n + \\
 & + w_6 u_{i,j-1}^n + w_7 u_{i+1,j+1}^n + w_8 u_{i+1,j-1}^n + w_9 u_{i-1,j+1}^n + \\
 & + w_{10} u_{i-1,j-1}^n + u_{i,j}^n - I_{i,j}^n
 \end{aligned} \tag{2.10}$$

The weights w_1, w_2, \dots can be any value and include diffusion coefficients, spatial step and time step. In the presence of fibrosis some of the elements can be equal to 0, thus making it possible to significantly vary the final solution. The weights are calculated before the model launch and then go to the main computational core of the package. Nevertheless, the functionality of Finitewave allows to recalculate the weights during the main computational loop.

2.4 Performance strategies

Writing high-performance programs in Python is difficult. The Python virtual machine only interprets the code without pre-optimizing it. In addition, Python disables multithreading, which makes it difficult to effectively parallelize programs. Fortunately, a large number of solutions to this problem have been proposed.

2.4.1 Numba

The main acceleration of numerical schemes in Finitewave on the CPU is implemented using Numba, which allows to JIT-compile the code and call it from the Python standard code, this also permits multithreading, increasing the execution speed by parallelizing the loops. In this case, part of the code must be moved outside the standard Python code for which kernels exist. The executable Numba code must be a separate function wrapped with the `@njit` decorator, which can also contain the `'parallel = True'` flag to enable multithreading. This special code section must meet the Numba requirements given in the documentation [7].

2.4.2 PyCUDA

Modern high performance computing would be unthinkable without the GPU, which has gained more popularity over the past decades. Its main idea is to split computational work among hundreds of small processors capable of performing basic mathematical operations with float numbers.

Finitewave has a separate subpackages (gpuwave2D and gpuwave3D) that allows to transfer the calculations to the GPU. Its logic is very similar to the logic of the CPU-package, the only difference is that the kernels are written using the pyCUDA library. PyCUDA allows to write the most optimized programs using CUDA C language and call it directly from the standard Python code, greatly speeding up the execution.

2.4.3 Vectorization

Most of the array operations in Finitewave are performed through Numpy arrays, which also creates a number of opportunities for code optimization. Numpy provides a programming interface for well-optimized C code to perform the necessary actions, which in some cases helps to avoid the use of Python loops. This procedure is called vectorization and consists in reducing all the code to operation with arrays only. With the help of vectorization, in particular, a number of trackers and part of the code that implements the weights calculation were implemented.

2.5 Example

Finitewave contains a lot of examples that will help to understand the basics of its work and cover all the main functions: mesh preparation, calculation, trackers and various stimulation protocols usage, postprocessing. All the examples are located in the root directory of the Finitewave repository. Here we consider one simplest example of running the excitation wave in 2D square medium with the size 100x100 nodes 2.7. Our task is to run the excitation wave and write the animation of the process as mp4 file.

The example starts with the tissue initialization. We create a 100x100 integer array to represent a mesh of conductive nodes surrounded by the boundary lines:

```
n = 100
tissue = CardiacTissue2D([n, n])
tissue.mesh = np.ones([n, n], dtype="uint8")
tissue.add_boundaries()
```

As we use an isotropic medium we don't add any information about the fibers.

Next we initialize a new electrophysiological model with TP06 class - one of the pre-implemented models ready to use in simulations. Three basic parameters

are required: time step (dt), spatial step (dr) and maximal calculation time (t_{max}):

```
tp06 = TP062D()
tp06.dt = 0.02
tp06.dr = 0.25
tp06.t_max = 300
```

To initiate the excitation wave we create a new `StimVoltageCoord2D` class object with the time of the stimulation (0 ms), voltage value which will be assigned to the selected region and the rectangular region of stimulation itself with the coordinated (0, 100, 0, 5). The excitation wave will be launched at the top side of the square mesh:

```
stim_sequence = StimSequence()
stim_sequence.add_stim(StimVoltageCoord2D(0, 1, 0, n, 0, 5))
```

The concept of trackers in `Finitewave` is to create a list of trackers that will be called every time step to complete their tasks. We are creating an animation tracker that will create snapshots of the potential with the selected step:

```
tracker_sequence = TrackerSequence()
animation_tracker = Animation2DTracker()
animation_tracker.target_array = "u"
animation_tracker.dir_name = "anim_data"
animation_tracker.step = 5 # every 5 ms
tracker_sequence.add_tracker(animation_tracker)
```

Add all the initialized objects to the model object and run calculations:

```
tp06.cardiac_tissue = tissue
tp06.stim_sequence = stim_sequence
tp06.tracker_sequence = tracker_sequence

tp06.run()
```

After the calculations are finished, now it is time to assemble the animation. For this, we use the additional package tools. In this case, the `AnimationBuilder`, which gets the path to the prepared frames:

```
# build animation with the ffmpeg utility
animation_builder = AnimationBuilder()
animation_builder.dir_name = "anim_data"
animation_builder.write_2d_mp4("animation.mp4")
```

We don't need the frames any more and can delete it:

```
shutil.rmtree("anim_data")
```

2.6 Chapter summary

Computer modeling relies on solid numerical methods, which can be challenging to implement. Specialized computational packages can solve this problem by allowing researchers to focus on their subject area. In the field of modeling in cardiology, there are also ready-made solutions for the problem of modeling the cardiac electromechanical activity, but their usage may require a long learning time. In addition, it can require a lot of computational resources. The software package developed by us is an attempt to solve this problem by providing researchers, students and other users with a lightweight and affordable tool for modeling the electrophysiology of cardiac tissue in 2D and 3D tissues. The numerical method is based on a weighted scheme that allows solving a wide range of problems, including modeling domains with complex boundaries, fibrosis of various patterns, as well as regions with low conductivity areas. The package also includes a set of out-of-the-box tools and provides the user with a large number of examples to get started with simulations. Our software can be applied for: teaching mathematical modeling in cardiology, fast hypothesis testing, fast dataset formation for further use in machine learning problems, tasks related to modeling fibrosis in large-scale models.

```
n = 100
tissue = CardiacTissue2D([n, n])
tissue.mesh = np.ones([n, n], dtype="uint8")
tissue.add_boundaries()

tp06 = TP062D()
tp06.dt = 0.02
tp06.dr = 0.25
tp06.t_max = 300

stim_sequence = StimSequence()
stim_sequence.add_stim(StimVoltageCoord2D(0,1,0,n,0,5))

tracker_sequence = TrackerSequence()
animation_tracker = Animation2DTracker()
animation_tracker.target_array = "u"
animation_tracker.dir_name = "anim_data"
animation_tracker.step = 5 # every 5 ms
tracker_sequence.add_tracker(animation_tracker)

tp06.cardiac_tissue = tissue
tp06.stim_sequence = stim_sequence
tp06.tracker_sequence = tracker_sequence

tp06.run()

# build animation with the ffmpeg utility
animation_builder = AnimationBuilder()
animation_builder.dir_name = "anim_data"
animation_builder.write_2d_mp4("animation.mp4")

shutil.rmtree("anim_data")
```

Figure 2.7: Code listing with the example of excitation wave launching and recording its animation. Packages import omitted for the shortness of the code.

References

- [1] P. Gernot, A. Loewe, A. Neic, C. Augustin, Y-L. Huang, M.A.F. Gsell, E. Karabelas, Jorge Sanchez Mark Nothstein, Anton J Prassl, Gunnar Seemann, and Edward J Vigmond. *The openCARP Simulation Environment for Cardiac Electrophysiology*. pages under review, bioRxiv preprint available, 2021.
- [2] F.R. Cooper, R.E. Baker, M.O. Bernabeu, R. Bordas, L. Bowler, A. Bueno-Orovio, H.M. Byrne, V. Carapella, L. Cardone-Noott, and B.D. Evans A.G. Fletcher J.A. Grogan W. Guo D.G. Harvey M. Hendrix D. Kay J. Kursawe P.K. Maini B. McMillan G.R. Mirams J.M. Osborne P. Pathmanathan J.M. Pitt-Francis M. Robinson B. Rodriguez R.J. Spiteri D.J. Gavaghan J. Cooper, S. Dutta. *Chaste: Cancer, Heart and Soft Tissue Environment*. J. Open Source Softw, page 1848, 2020.
- [3] Michael Clerx, Pieter Collins, Enno de Lange, and Paul G. A. Volders. *Myokit: A simple interface to cardiac cellular electrophysiology*. Progress in Biophysics and Molecular Biology, 120:100–114, 2016.
- [4] P. Chinchapatnam, K.S. Rhode, M. Ginks, C.A. Rinaldi, P. Lambiase, R. Razavi, S. Arridge, and M. Sermesant. *Model-based imaging of cardiac apparent conductivity and local conduction velocity for diagnosis and planning of therapy*. IEEE Trans Med Imaging, 27:1631–42, 2008.
- [5] H.P. Langtangen and S. Linge. *Finite Difference Computing with PDEs - A Modern Software Approach*. Springer International Publishing, 2017.
- [6] R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. SIAM, 2007.
- [7] S.K. Lam, A. Pitrou, and S. Seibert. *Numba: a LLVM-based Python JIT compiler*. Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, pages 1–6, 2015.