Benchmarking and Optimization of Cardiac Electrophysiology Solvers

Eina Bergem Jørgensen

Thesis submitted for the degree of
Master of Science
60 credits

Department of Physics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2021

Benchmarking and Optimization of Cardiac Electrophysiology Solvers

Eina Bergem Jørgensen

Benchmarking and Optimization of Cardiac Electrophysiology Solvers

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

In this thesis we compare two software codes used in cardiac electrophysiology research, *cbcbeat* [Marie Rognes et al., 2017] and *openCARP* [Plank et al., 2021b], in terms of efficiency. The comparison is done by reproducing the [Niederer et al., 2011] benchmark to verify the accuracy of the implementations, and performing measurements of the wall time spent by the solvers when computing a simulation. The time is measured for the different components of the solvers, and for various temporal and spatial resolutions. Both solvers use operator splitting which divides the solution process into an ODE step and a PDE step, and were implemented to use the same numerical methods.

The efficiency benchmark showed that openCARP outperforms cbcbeat by 8-9 times when run in serial execution. cbcbeat appears to benefit more from parallelization than openCARP, at least for a problem of the size tested in this thesis, with openCARP being about 2.5-3 times faster when run in parallel. Furthermore, the detailed timing of the various solution steps, indicate that cbcbeat's total computation time is dominated by a sub-optimal ODE solver, that spends a lot of time computing the ion current term of the monodomain equation, as well as being subject to overhead when calculating the integration steps. The ODE solver in openCARP accounts for only a quarter of the total wall time and is almost 20 times as fast as cbcbeat when timed for serial execution. The PDE solver in both cbcbeat and openCARP utilized PETSc [Abhyankar et al., 2018] well, and is probably as good as optimized. When assembling the right hand side vector of the linear PDE, however, openCARP only spends $\sim$15 % of the time cbcbeat does on the same operation, making it evident that the assembly functionality in cbcbeat leaves room for improvement. Our benchmark reveals that targeting the aforementioned components of cbcbeat for optimization, could have the potential to drastically improve its efficiency.

Acknowledgements

This thesis could not have been written, had it not been for several people who all deserve a heartfelt thank you.

First and foremost, I would like to thank my supervisor Joakim Sundnes, who introduced me to the project, and who's excellent guidance, advice and encouragement have been invaluable throughout the work with this thesis.

Thank you to Morten Hjorth-Jensen and the rest of the people at CCSE, for creating a wonderful environment in which to do a master's degree, both academically and socially. I wish that the covid-19 pandemic had not prevented me from taking more advantage of it.

I would also like to thank Cécile Daversin-Catty and Hermenegild Arevalo for sharing your resources and knowledge of cbcbeat and openCARP with me.

Finally, I want to thank my wonderful friends and family for your never ending encouragement and support.

Oslo, 31.5.2021

Eina Bergem Jørgensen

Contents

List of Figures

All figures for which a source is not cited, are created by the author.

List of Tables

Chapter 1

Introduction

Heart disease is the the leading cause of death in the western world [Virani et al., 2021], and is such a research field of great interest. This thesis will delve into the physiologically complex phenomenon that is the electrophysiology of the ventricular myocyte. The majority of sudden cardiac arrests are caused by ventricular arrhythmias [Zipes and Wellens, 2000]. During ventricular fibrillation, disorganised electrical activity causes the heart to be unable to pump normally. It is fatal if not treated immediately, with a survival rate of out-of-hospital cases at only 17 % [Berdowski et al., 2010]. By creating accurate simulations of the electrical activity in the heart, one can both study the cause of ventricular fibrillation and other heart conditions, as well as the remarkable effect of defibrillation or other cardio-vascular disease treatments.

In the field of computational science, mathematical models are often used to describe physical phenomenons, which can then be numerically solved. With the continuous development of numerical methods and computational power, this has become a powerful scientific tool. It allows us to study models in detail, and perform numerical experiments without or with reduced need of laboratory work. Biological systems are amazingly complicated, with many components interacting in just fractions of a second. The mathematical models involved in describing their behavior often consist of complex coupled equations that are impossible to solve analytically. Cardiac physiology is therefore a field that has greatly benefited from the rise of computational science.

With this type of modelling and simulations, there will always be a trade off between the efficiency of the computations and the physical or physiological accuracy of the results, both in terms of the choice of complexity in the system of equations

to be solved, and in the solution method used.

1.1 Motivation

Modern cardiac electrophysiology models are complex, consisting of systems of coupled partial differential equations and non-linear ordinary differential equations. Due to the rapidly changing electric potentials in the heart, gradients are steep, and high spatial and temporal resolution is required to achieve accurate numerical solutions. With time steps as small as 0.005 ms and a spatial resolution up to 0.1 mm, simulations for only a small slice of heart tissue can have hundreds of thousands of degrees of freedom. The models are therefore computationally expensive, causing computation time to be a considerable bottleneck for research.

Different research groups around the world have developed their own software for computing simulations used for cardiac research. In 2011, eleven different cardiac tissue electrophysiology software codes were compared in an N-version benchmark presented by [Niederer et al., 2011]. The goal was to create a standardized problem, and by comparing solvers, find a gold standard converged solution that can be used for code verification.

The Niederer benchmark did not include metrics such as computation wall time, as the main focus was a verification of accuracy. It is however something that could be of great value, as computation time, as stated, can be a severe bottleneck for model based research.

In this thesis we examine two different code platforms used for solving cardiac tissue electrophysiology simulations, that were both part of the Niederer benchmark. The solvers in question are:

- *cbcbeat* [Marie Rognes et al., 2017], which is developed and used by Center for Biomedical Computing hosted by Simula Resarch Laboratory.
- *openCARP*[Plank et al., 2021b], which development started at the University of Graz and Liryc in Bordeaux, and is used by several groups around Europe doing cardiac electrophysiology research.

They are two softwares that are relevant to compare, as they utilize a lot of the same methods, such as operator splitting, finite element methods and the use of

PETSc [Abhyankar et al., 2018] for solving the differential equations. They are also both used in active research at Simula.

According to users, openCARP is known to be faster, but it is not known how much faster or what exactly is causing it to be faster. Quantifying the computational cost by performing a detailed timing of the two solvers for a standardized problem, would allow us to identify which parts of the solution process that is most time consuming, and how efficiently they are handled in the different implementations. Additionally, it is not known how the two solvers compare regarding parallelization, which is knowledge that could be beneficial to have.

Information attained from such benchmarks can provide valuable insight both in terms of choosing cardiac electrophysiology software that is both accurate and efficient, and for potentially improving the efficiency of cbcbeat.

1.2 Goals of the Thesis

The thesis has two main goals:

Firstly, we want reproduce the Niederer benchmark in both cbcbeat and openCARP. To achieve this we must familiarize ourselves the mathematical models and numerical methods needed solve the benchmark case numerically, as well as the two solvers and their implementation. By comparing our converged results to the results from the benchmark we can verify that we have implemented the case correctly in both solvers, which is crucial if the measurement of computation time is to have any value. As both solvers have been under development the last ten years since the benchmark, we are also interested in whether the convergence rate of the solvers have changed, as this was something that varied greatly between the 11 solvers in the original Niederer benchmark.

Secondly, we want to do a thorough benchmarking of the two solvers' efficiency. The aim is to quantify the difference in computational cost by measuring the time spent on simulating the given case. We are both interested in the total computational wall time, and the time spent on the different operations within the solver that make up the simulations. We wish to study which parts of the solvers that contribute to the difference in efficiency, by doing detailed measurements of the wall time for the simulations, for all combinations of the spatial resolutions 0.5, 0.2, and 0.1 mm, and temporal resolutions 0.05, 0.01 and 0.005 ms. We also want to identify how cbcbeat and openCARP behave under parallelization, by running

the simulations for a selection of resolutions using an increasing number of parallel threads.

Hopefully, the results provided by our benchmark in combination with the insight gained into the two solvers' architecture, will allow us to identify areas of potential improvement in cbcbeat that could increase its efficiency.

1.3 Chapter Overview

Chapter 2 gives an introduction to the physiology of the heart and its properties, as well as present mathematical models used to describe cardiac tissue behavior. We will go through the type of equations included in such models, and the schemes and methods used to solve them numerically.

In chapter 3 we will present the two solvers cbcbeat and openCARP. We will cover the architecture of the solvers, and examine the differences and similarities in how they implement the numerical methods. The chapter also covers the details of the Niederer benchmark case as well as how it is coded in both solvers, and how the implementation is verified and timed.

Chapter 4 presents the results of our simulations. First we present our results reproducing the Niederer benchmark for verification of our code. Then we present the results of the time benchmark.

Finally the results and their implications are discussed in chapter 5, and we make some conclusive remarks in chapter 6

Chapter 2

Background Theory

This chapter covers the background theory that is relevant to the thesis. We present the cardiac physiology, mathematical models, and numerical and computational theory that make up the basis for the simulations that are to be examined.

2.1 Cardiac Physiology

Understanding the underlying behaviour, physics, and in this case physiology, of the objects and situations that we want to create simulations of is essential if the results of our computations are to be meaningful and verifiable. This section is an introduction to the relevant cardiac physiology and is for the most part based on *Mathematical physiology* [Keener and Sneyd, 1998d] and *Physiology of the heart* by [A. M. Katz, 2011a].

2.1.1 The Heart

The case we are going to study in depth in this thesis, is a simulation on a small 3D slice of heart tissue. It is however in our interest to have some context regarding how the heart functions as a whole, and we will therefore cover it briefly.

The human heart consists of four chambers, and can be viewed as two pumps that operate in series. The right atrium and the right ventricle receive deoxygenated blood from the systemic veins and pumps it through the lungs in the pulmonary

circulation. The left atrium and ventricle receive oxygenated blood from the lungs and pumps it to the body in the systemic circulation [A. M. Katz, 2011b]. On each side, the blood enters the heart through the atria, and is further pumped into the ventricle. It is prevented from flowing backward from the ventricle to the atria through the atrioventricular valves. In figure 2.1, an illustration of the blood flow through the different chambers of the heart is presented.

The cardiac muscle cells, called the myocardial cells or cardiomyocyte, are both excitable and contractile [Keener and Sneyd, 1998e]. That is, they can both conduct an electrical signal along their membrane, and physically contract. The contraction of the cardiomyocytes is the driving force pumping the blood.

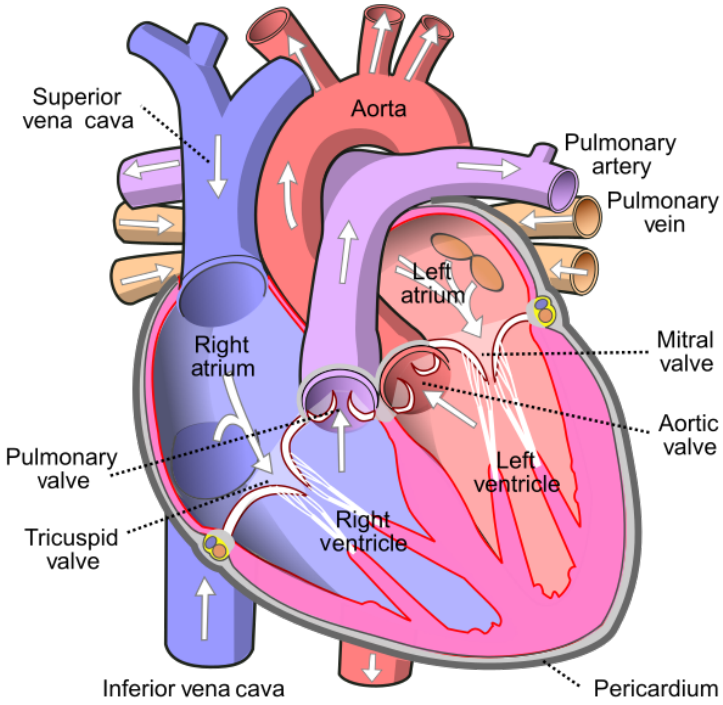


Figure 2.1: Illustrative diagram of the human heart, showing its main components of chambers, veins and valves. The direction of the blood flow is indicated by the white arrows. Illustrated by Eric Pierce via Wikimedia Commons, licensed under CC BY-SA 3.0 [CreativeCommons, 2021b]

The heartbeat is initiated and controlled by electrical impulses generated by the sinoatrial node (SA node), which is the primary pacemaker site within the heart.

It is located at the top of the right atrium below the superior vena cava, and generates electrical pulses. The electrical signal is called the *action potential*, and is propagated through the atria along the myocardial cells, causing the atria to contract. The electrophysiology of heart tissue's reaction to the electrical pulse will be discussed more in detail in section 2.1.4.

When the action potential reaches the bottom of the atria, the conduction along the myocardial cells is stopped by a layer of non-excitable cells, hindering the signal from propagating to the ventricle. The only path for the signal to take is through the atrioventricular node, which leads the signal to the bottom of the ventricle through the bundle of HIS, made up of Purkinje fibers. Exiting the Prukinje fibers, the signal is then free to propagate upwards along the ventricular myocardial cells, causing a second contraction, this time in the ventricle. A diagram of the conductive system of the heart is presented in figure 2.2.

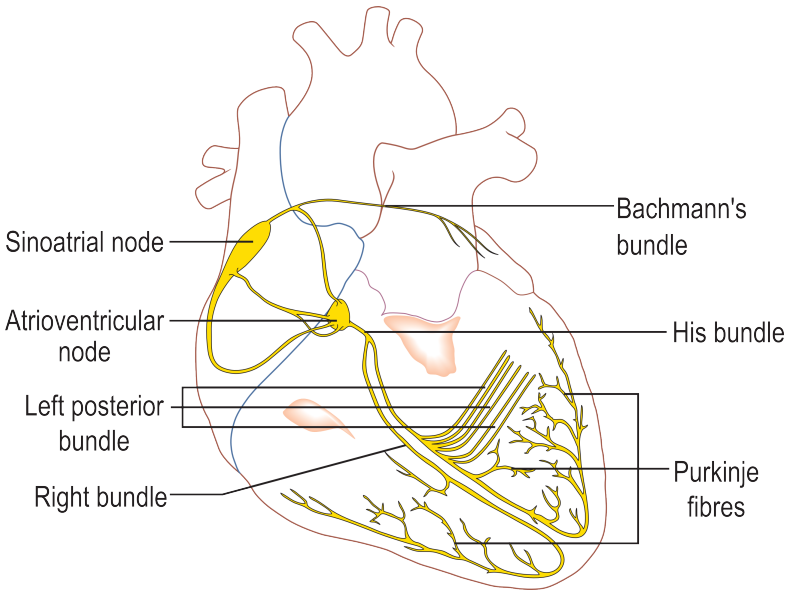


Figure 2.2: Illustrative diagram of the electrical conduction system of the human heart. Illustrated by Wikimedia Commons user Madhero88 via Wikimedia Commons, licensed under CC BY-SA 3.0

In order to understand the details of the conduction of the action potential and the contraction of the myocardium, we need to understand the architecture of the cardiac cell membrane.

2.1.2 The Cardiac Cell Membrane

The cardiac cell membrane is, like other biological membranes, made up of a double layer of phospholipids which gives the membrane a hydrophobic surface on each side, and a hydrophilic core. The intracellular and extracellular environments contains dissolved salts, mainly NaCl and KCl. These salts dissolve to Na^+, K^+ and Cl^- ions. The non-polar surface of the cell membrane acts as a barrier, making the membrane impermeable to charged ions [A. M. Katz, 2011b]. Since the membrane acts as an insulator between the electrically charged ions on each side of it, we can think of it as a capacitor. There will therefore be an electric potential across the cell membrane, which magnitude is dependant on the charge and concentration of ions in the intracellular and extracellular space. This potential is called the *transmembrane potential*, and it gives rise to a capacitive current I_{cap}, proportional to the capacitance C_m of the membrane. Figure 2.3b illustrates this.

$$C_m \frac{dV}{dt} = I_{\text{cap}} \tag{2.1}$$

Proteins imbedded in the bilayer are responsible for many important activities in the cell. The most relevant in our case are the protein lined pores, called *channels*. These channels can allow for passage of certain charged molecules, that would otherwise not be able to move through the non-polar membrane surface. They have the ability to be ion specific and can open and close based on certain conditions, so that for example sodium or potassium ions only can pass through some channels, and only in given situations [Keener and Sneyd, 1998b]. This allows for ions to be transported through the membrane and change or maintain their intracellular and extracellular concentrations. A simple illustration of the cell membrane with phospholipids and a protein channel is presented in figure 2.3a.

If charged ions are flowing in and out of the membrane, this is denoted as the *ion current* I_{ion}, and the total transmembrane current is the sum of the capacitive and ion current.

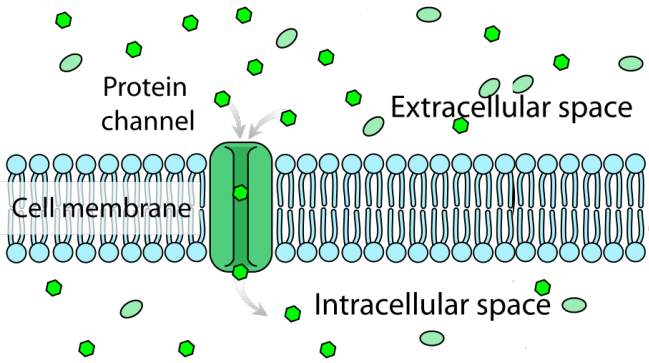
$$I_\tau = I_{\text{cap}} + I_{\text{ion}} = C_m \frac{dV}{dt} + I_{\text{ion}} \tag{2.2}$$

In order for there not to be a build up of charge on either side of the membrane over time, the sum of the currents must be zero, which gives

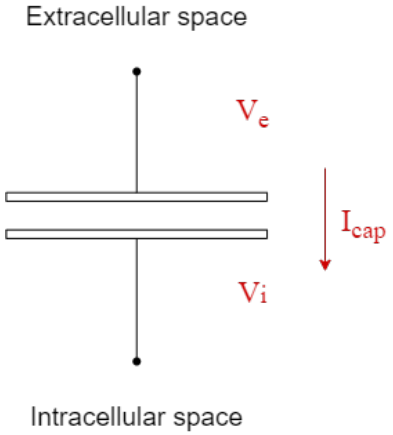
$$C_m \frac{dV}{dt} + I_{\text{ion}} = 0 \tag{2.3}$$

or in the case of an applied simulus current

$$C_m \frac{dV}{dt} = -I_{\text{ion}} + I_{\text{stim}} \tag{2.4}$$



(a) Simplified illustration of the cell membrane. The cell membrane is made of a bilayer of phospholipids, which gives a non-polar membrane surface impermeable to charged ions. Protein channels in the membrane allow for passage of certain charged ions under given conditions. Figure adapted from illustration by Mariana Ruiz Villarreal via Wikimedia Commons, licensed under CC0 [CreativeCommons, 2021c].



(b) The transmembrane potential illustrated with a circuit diagram. The cell membrane works as an insulator between charged ions in the intra- and extracellular space, and a difference in charge leads to a potential $V = V_i - V_e$.

Figure 2.3

2.1.3 Ionic Currents and the Nernst-Planck Equation

Assuming that a channel for a given ion is open, and that they are free to move in and out of the cell, the passive transport flux of such ions will be a result of two physical phenomenons. Firstly, a difference in ion concentration in the intracellular and extracellular environment will lead to diffusion. *Frick's law* states that the diffusive flux is proportional to the magnitude of the concentration gradient:

$$\boldsymbol{J}_{\text{diff}} = -D\boldsymbol{\nabla} c \tag{2.5}$$

where $\boldsymbol{\nabla} c$ denotes the concentration gradient, and the scalar D is the diffusion coefficient, all for a certain ion.

Secondly, electrodynamics come into play as the electric field from the transmembrane potential discussed in the previous section will exert a force on any charged ions, depending on the strength and sign of the transmembrane potential. This leads to a flux due to electric drift that is given by *Planck's equation*:

$$\boldsymbol{J}_{\text{drift}} = \frac{-DzF}{RT} c \boldsymbol{\nabla} V \tag{2.6}$$

where D again is the diffusion coefficient, z is the valance of the ion, F is the Faraday constant, R is the gas constant, T is the temperature in Kelvin, c is the concentration and $\boldsymbol{\nabla} V$ is the electric field, which is the gradient of the membrane potential.

Combining (2.5) and (2.6) gives the total flux $\boldsymbol{J}$ described by the *Nernst-Planck equation*:

$$\begin{aligned} \boldsymbol{J} &= \boldsymbol{J}_{\text{diff}} + \boldsymbol{J}_{\text{drift}} \\ \boldsymbol{J} &= -D\left(\boldsymbol{\nabla} c + \frac{zF}{RT} c \boldsymbol{\nabla} V \right) \end{aligned} \tag{2.7}$$

The two components contributing to the total flux can either work in the same direction or opposite one another, depending on the concentration of the ion and the strength and direction of the electric field. It is the nature of such a relationship, that there exists an equilibrium in which the total flux across the membrane is zero. By setting $\boldsymbol{J} = 0$ in a one dimensional version of (2.7), we can find the expression for this equilibrium. We write

$$\boldsymbol{J} = -D\left(\frac{\partial c}{\partial x} + \frac{zF}{RT} c \frac{\partial V}{\partial x} \right) = 0 \tag{2.8}$$

so that

$$\frac{1}{c} \frac{\partial c}{\partial x} = -\frac{zF}{RT} \frac{\partial V}{\partial x} \tag{2.9}$$

an by taking the integral over the membrane with width L using $x \in [0, L]$ with c_i, V_i at $x = 0$ and c_e, V_e at $x = L$ we get

$$\int_0^L \frac{1}{c}\frac{\partial c}{\partial x}dx = -\int_0^L \frac{zF}{RT}\frac{\partial V}{\partial x}dx \tag{2.10}$$

so that

$$ln(c)\Big|_{c_i}^{c_e} = \frac{zF}{RT}(V_i - V_e) \tag{2.11}$$

which leads to

$$V = \frac{RT}{zF}\ln\left(\frac{c_e}{c_i}\right) \tag{2.12}$$

where $V = V_i - V_e$. (2.12) is the *Nernst-potential*, and gives the transmembrane potential for each ion at which the net flux in and out of the cell is zero. It is often denoted as E_X, where X is an ion, i.e. E_{NA} is the Nernst equilibrium potential of sodium and will be the potential that the sodium current will push the transmembrane potential towards. This allows us to know the direction of the ion current for each ion based on their intracellular and extracellular concentration, and the current membrane potential.

As an example, both sodium and potassium have valence $z = 1$ while calcium has valence $z = 2$, and the typical values for their concentration and following Nernst potential in the mammalian myocytes are given in table 2.1.

ion	c_e	c_i	Nernst Potential given by (2.12)
Na^+	110 mM	8 mM	70 mV
K^+	4 mM	100 mM	-86 mV
Ca^{2+}	1 mM	0.0002 mM	114 mV

Table 2.1: Average values for intracellular and extracellular ion concentration in mammalian myocytes and their respective Nernst equilibrum potential. Concentration values from [A. M. Katz, 2011c].

If the membrane potential is lower than 70 mV and higher than -86 mV the sodium and calcium currents will go inwards in order to try and increase the membrane potential towards their Nernst-equilibrium, while the potassium current will go outwards to decrease it, given that their channels are open so the ions are free to

flow through the cell membrane. This then leads us to the question: when are they open?

The relationship between the strength of a current and the magnitude of the transmembrane potential, can be described by a so called current-voltage curve, or $I-V$ curve. The shape of the curve will reflect underlying biophysical mechanisms in the cell, and can vary in different cell types. Hodgkin and Huxley determined experimentally in 1952, that for a squid giant axon this relationship is approximately linear [Hodgkin, Huxley, and B. Katz, 1952]. If we assume that to be the case, the current for an ion X is given by

$$I_X = g_X(V - E_X) \tag{2.13}$$

where g_X is the conductance of the ion through the channel, V is the transmembrane potential, and E_X is the Nernst-equilibrium. The value of g_X must therefore encompass the state of the channels in the membrane, as closed channels will give low conductance, while open channels will give higher conductance. Hodgkin and Huxley described a model to explain this process [Hodgkin and Huxley, 1952], for which they won the 1963 Nobel Prize in medicine. We will go into further detail about the mathematics of the model in section 2.2.1, but first we will take a look at a qualitative understanding of the ion currents as the ventricular myocyte is exposed to the action potential.

2.1.4 The Action Potential and Channel Gating

Studying a point at the membrane of the ventricular myocyte, the graph of the membrane action potential over time has a distinct shape, and lasts relatively long, compared to i.e. a skeletal action potential. This is displayed in figure 2.4. At rest, the cell membrane is most permeable to potassium, leading the resting potential to be around -80 mV, which is close to the Nernst potential of K^+. [A. M. Katz, 2011c].

Apart from *at rest* (phase 4), the ventricular cardiac action potential is generally categorized into four other phases. During *the upstroke* (phase 0) the membrane depolarizes in a rapid positive increase of the membrane potential. This happens when an initial depolarizing stimuli is of an amplitude larger than a certain threshold, which triggers the opening of sodium channels. Na^+ then flows into the cell, increasing the potential. These sodium currents I_{Na} are often called the *fast inward currents*.

As the sodium channels are inactivated, potassium channels will open and close

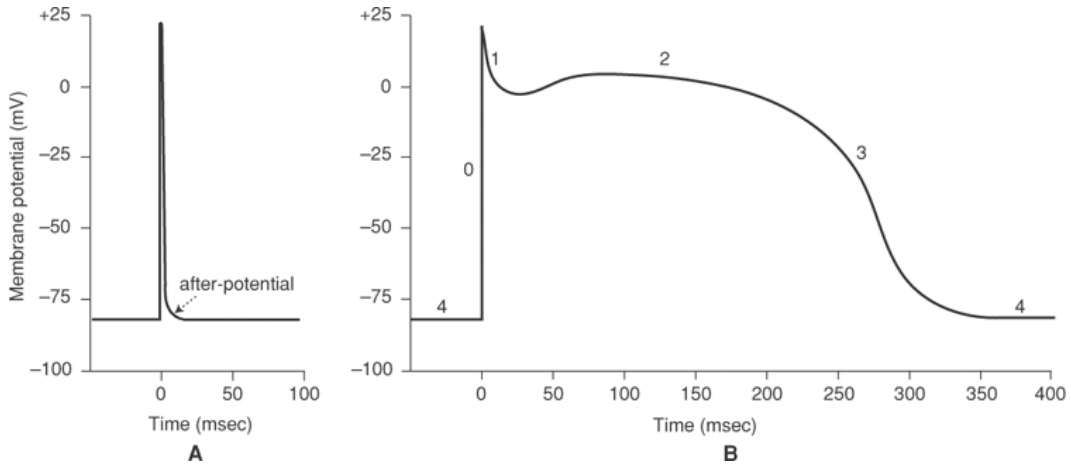


Figure 2.4: [A] The skeletal muscle action potential is a simple short spike in membrane potential. [B] The cardiac muscle action potential in the ventricle lasts over 300 ms, and has a distinct shape given by its five phases: *rest/diastole* (4), *upstroke* (0), *early repolarization* (1), *plateau* (2), and *repolarization* (3). Figure taken from [A. M. Katz, 2011c], with permission.

rapidly, allowing the the *transient outward currents* I_{to}, to cause *early repolarization* (phase 1). Then follows the *plateau* (phase 2), where L-type calcium channels activated by the initial sodium current opens so that Ca^{2+} flows into the cell with the I_{CaL} current. This in turn activates chloride channels, and the inward Ca^{2+} and Cl^{-} currents, in combination with the outward K^{+} current, balance out leaving the membrane potential relatively unchanged for a period.

It is the increase in intracellular calcium concentration that causes the contraction of the cardiac cell in the so called excitation-concentration coupling [Keener and Sneyd, 1998a]. The intracellular space, the *myoplasm*, contains the sarcoplasmic reticulum (SR) which primary function is to store calcium, and release or receive it from the myoplasm. When Ca^{2+} enters the cell, the SR will release more calcium from the *ryanodine receptors* in a process called calcium- induced calcium release (CIRC). The calcium will bind to the filaments in the myocyte, causing it to contract. As it is not the subject of this thesis, we will not elaborate further on the contractile mechanisms of cardiac tissue, but rather its conductive qualities.

Eventually, the L-type calcium channels close, and the *delayed rectifier currents* I_{K} which is a net outward current of positive K^{+} ions, allows for *repolarization* (phase 3), and the membrane potential returns to its resting value. Back at rest (phase 4), the inward rectifier current I_{K1}, the Na^{+}-Ca^{2+} exchanger I_{NaCa} and Na^{+}-K^{+} ATPase I_{NaK} transports potassium into the cell and calcium out of the cell or into

the SR. This restores the ion concentration values and relaxes the myocyte, thus preparing the cell for a new action potential and contraction, all while keeping the membrane potential constant.

We now have an overview of the mechanisms of the opening and closing of the ion channels in the membrane, which gives the action potential of the ventricular myocyte its characteristic shape. It is however a simplified explanation, as there are many details regarding the both the ion currents and the intracellular calcium dynamics that have been omitted. For instance, both I_{K} and I_{to} have been discovered to consist of several components, for example fast-activating (I_{Kur}, I_{Kr}, $I_{\mathrm{to,f}}$), slow-activating (I_{Ks}, $I_{\mathrm{to,s}}$) or calcium sensitive currents (I_{to2}, $I_{\mathrm{K.\ Ca}}$) [Wang et al., 1994], [Xu et al., 1999]. In the next section, we will take look at how different mathematical models over time have developed to include the more detailed dynamics of these currents.

2.2 Mathematical Models

When developing mathematical models to represent physical phenomenons, the aim is to accurately describe the behaviour of a system so that it can be studied. Simultaneously, more complex models come with computational cost, which has to be taken into account. Depending on what we aim to study, it might be beneficial to represent certain mechanisms in great detail, and others more superficially, with simpler equations that mimic their observable general behaviour. In the upcoming sections we will cover how the qualitative understanding of how the action potential works has been translated to mathematical models that can be used to recreate its behaviour and perform numerical experiments. We will briefly cover the historic development of cardiac electrophysiology modelling, and discuss a few models for the cardiac cell with different levels of complexity. Additionally, we will introduce the bidomain and monodomain model for conduction along the cell membrane, to give us the complete set of equations that need to be implemented for the simulations we are to use as the case for our benchmark.

2.2.1 The Hodgkin-Huxley Model

In 1952 Hodgkin and Huxley published a series of articles in the *Journal of Physiology* in which they presented their research successfully measuring ion currents in the giant squid axon and introduced a quantitative description of the ion currents

conductance [Hodgkin and Huxley, 1952]. Their theory and model is regarded as an important landmark in physiology [Keener and Sneyd, 1998c]. In order to make the more complex model used in our numerical simulations more understandable, we first introduce the ideas presented by Hodgkin and Huxley, on which modern cell models are built.

In the previous section we covered a qualitative understanding of what happens to channels and ion currents in the membrane during an action potential, as well as some common names for some of the different currents that have been discovered and added to the theory as the field of electrophysiology has been researched extensively over the last seventy years. In Hodgkin and Huxley's model, they included 3 currents, the Na^+ current, the K^+ current, and the *leakage current* (see figure 2.5). The latter is a combination of several small currents, most notably the Cl^- current, lumped into a single expression. The total ion current is given as:

$$I_{\text{ion}} = I_{\text{Na}} + I_{\text{K}} + I_{\text{l}} \tag{2.14}$$

For each current, a linear $I - V$ curve as in (2.13) is used

$$I_{\text{ion}} = g_{\text{Na}}(V - E_{\text{Na}}) + g_{\text{K}}(V - E_{\text{K}}) + g_{\text{l}}(V - E_{\text{l}}) \tag{2.15}$$

where g_* and E_* are the current specific conductance and Nernst potential, and V is the membrane potential.

Combining (2.4) and (2.15) gives the Hodgkin-Huxley (HH) model for cardiac cell, described by the equation:

$$C_m \frac{dV}{dt} = -g_{\text{Na}}(V - E_{\text{Na}}) - g_{\text{K}}(V - E_{\text{K}}) - g_{\text{l}}(V - E_{\text{l}}) + I_{\text{stim}} \tag{2.16}$$

In the Hodgkin and Huxley model, opening and closing of the different channels at various points during the action potential are handled by expressing the conductance in terms of a constant and so called secondary (or gating) variables, that satisfy a first order differential equation. For the potassium conductance, a single variable n, is used

$$g_{\text{K}} = \overline{g}_{\text{K}} n^4 \tag{2.17}$$

while the potassium conductance uses a combination of the faster gating variable m and the slower h

$$g_{\text{Na}} = \overline{g}_{\text{Na}} m^3 h \tag{2.18}$$

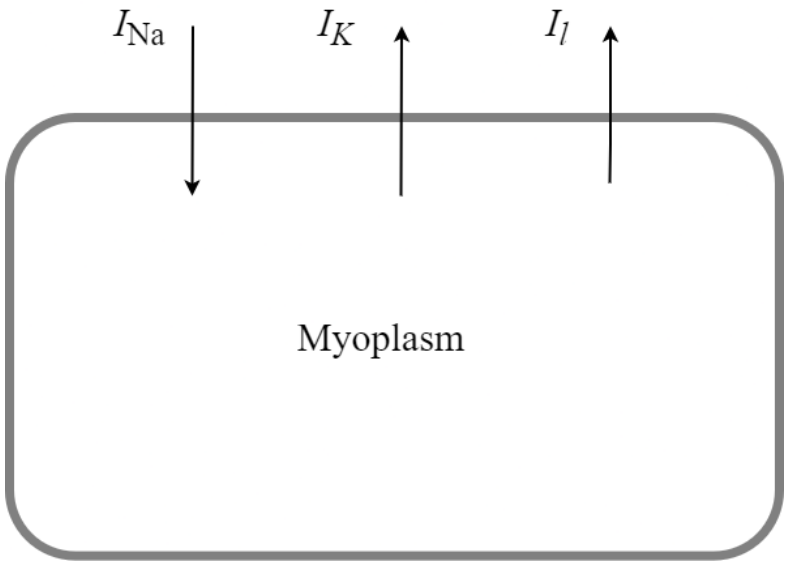


Figure 2.5: A diagram of the Hodgkin-Huxley cell model. The original HH model includes three currents, the sodium current I_{Na}, the potassium current I_{K}, and the leakage current I_{l}.

where $\overline{g}_{\mathrm{K}}$ and $\overline{g}_{\mathrm{Na}}$ are constants, and m, n and h satisfy the following differential equations:

$$\frac{dm}{dt} = \alpha_m(1-m) - \beta_m m \tag{2.19}$$

$$\frac{dn}{dt} = \alpha_n(1-n) - \beta_n n \tag{2.20}$$

$$\frac{dh}{dt} = \alpha_h(1-h) - \beta_h h \tag{2.21}$$

Hodgkin and Huxley obtained the functions α_m, α_n, α_h, β_m, β_n and β_h by fitting them to curves of measured experimental data. They are in units of $(\mathrm{ms})^{-1}$. The leakage current is given simply by a constant conductance $g_{\mathrm{l}} = \overline{g}_{\mathrm{l}}$.

The result is that (2.16) can be written as

$$C_m \frac{dV}{dt} = -\overline{g}_{\mathrm{Na}} m^3 h(V - E_{\mathrm{Na}}) - \overline{g}_{\mathrm{K}} n^4 (V - E_{\mathrm{K}}) - \overline{g}_{\mathrm{l}}(V - E_{\mathrm{l}}) + I_{\mathrm{stim}} \tag{2.22}$$

with m, n, and h as in (2.19) - (2.21) and $\mathrm{C}_m = 1\ \mu\mathrm{F/cm}^2$. For the conductances Hodgkin and Huxley used, in units $\mathrm{mS/cm}^2$:

$$\overline{g}_{\mathrm{Na}} = 120 \qquad \overline{g}_{\mathrm{K}} = 36 \qquad \overline{g}_{\mathrm{l}} = 0.3 \tag{2.23}$$

This way of modelling cardiac cells using ionic currents and conductances with gating variables are called the Hodgkin-Huxley type formulations. Different versions of these formulations have been applied to create numerous models for different cardiac cells over the years. Early examples are [Beeler and Reuter, 1977] for ventricular cells, [McAllister et al., 1975] for Purkinje fibres, and [Yanagihara et al., 1980] for the sinoatrial node.

Comparing the HH model to the many currents and dynamics mentioned in section 2.1.4, it is clear that their model is a relatively simplified representation of the physiology of the myocyte. Modern cell models have become increasingly complex and specialized. Regardless, the underlying mathematical approach introduced by Hodgkin and Huxley, is still used. An example could be the LR-II model of the ventricular cell [Luo and Rudy, 1994], which contains nine currents in and out of the cell, in addition to two pumps, the Na^{+}-Ca^{2+} exchanger, and four intracellular currents related to the calcium dynamics in the sarcoplasmic reticulum. Even more complex models exists, and when doing computational experiments one must try to find a balance between a model's physiological detail and accuracy, and the practicality of implementation given mathematical complexity and computational cost. A large library of mathematical electrophysiology models can be found at `https://cellml.org`, as a part of the CellML project [Cuellar et al., 2003]

2.2.2 The Aliev-Panfilov Model

To include accurate descriptions of the details of the underlying mechanisms in the heart, relatively complex cell models are required, such as the one we will discuss in the next section. Sometimes however, if computational resources are limited or for other reasons, we might be interested in a much simpler model. The properties one would want out of such a model is generally that it just mimics the shape of the pulse and restitution properties of the action potential in the myocardium with good precision, without taking into too much consideration all the underlying physiological phenomenons. An example of such a model is the Aliev-Panfilov model [Aliev and Panfilov, 1996].

Instead of multiple different gating variables in addition to V, which all contribute with their own differential equation, the Aliev-Panfilov (AP) model uses a single term v. The equations are given on a dimensionless form using u and τ, which are

dimensionless versions of the membrane potential V and the time t, where

$$V = 100 \text{ mV} \cdot u - 80 \text{ mV} \qquad t = 12.9 \text{ ms} \cdot \tau \tag{2.24}$$

The ion current is dependant on u and v, and is described as

$$I_{\text{ion}} = ku(u-a)(u-1) + uv \tag{2.25}$$

where $k = 8$ and $a = 0.15$ are constants, and v satisfy the differential equation

$$\frac{dv}{d\tau} = \left(\varepsilon_0 + \frac{\mu_1 v}{u + \mu_2}\right)(v + ku(u - a - 1)) \tag{2.26}$$

where $\varepsilon = 0.002$, $\mu_1 = 0.2$ and $\mu_2 = 0.3$.

This model only requires the solution of one differential equation, belonging to v, as opposed to HH, where the three variables m, n and h each satisfy a different ODE, or as we shall see shortly, more complex cell models with far more variables than that.

2.2.3 Ten Tusscher & Panfilov

The ten Tusscher and Panfilov 2006 (TT2) model for the ventricular cardiac cell is a modern model based on experimental measurements of human action potential duration restitution [Ten Tusscher and Panfilov, 2006], developed to study ventricular fibrillation. The model was created as a newer version of the [Tusscher et al., 2004] model, with a more detailed description of the intracellular calcium handling. The TT2 epicardial model is used by [Niederer et al., 2011] in their benchmark, and is such the model that is implemented in this thesis. A more detailed description of the specific parameters, constants, and their units will be explained in section 3.1 in chapter 3, but we will present the qualitative description and the mathematical equations of the model here.

The TT2 model includes the major ion currents mentioned in the overview of the action potential in in section 2.1.4, as well as well as intracellular calcium dynamics. The TT2 model uses a somewhat more sophisticated explanation for the calcium-induced calcium release (CIRC) than presented earlier. It includes a so called diadic subspace (SS), which is an area in the myoplasm where there is a very small distance between the SR and the cell membrane. Here, the ryanodine

Currents in the TT2 cell model	
Transmembrane Ion Currents	
I_{Na}	sodium current
I_{K1}	inward rectifier potassium current
I_{to}	transient outward current
I_{Kr}	rapid delayed rectifier current
I_{Ks}	slow delayed rectifier current
I_{CaL}	L-type calcium current
I_{NaCa}	sodium-calcium exchanger current
I_{NaK}	sodium-potassium pump current
I_{bNa}	background sodium current
I_{bCa}	background calcium current
I_{pK}	plateau potassium current
I_{pCa}	sarcolemmal calcium pump current
Intracellular Calcium Currents	
I_{rel}	calcium-induced calcium release current
I_{up}	sarcoplasmic reticulum (SR) calcium pump current
I_{leak}	SR calcium leak current
I_{xfer}	diffusive calcium current between diadic subspace and bulk cytoplasm

Table 2.2: The currents included in the TT2 model, with their notations as presented in [Ten Tusscher and Panfilov, 2006]. See figure 2.6 for a diagram of the cell model and the direction of the different currents and pumps.

receptors sense when the Ca^{2+} concentration increases, and releases more calcium into the SS with the I_{rel} current. The high calcium concentration in SS leads to diffusion I_{xfer} into the rest of the myoplasm. Calcium pumped back into the SR is done through the I_{up}-current, and a small leakage from the SR to the myoplasm is included in the I_{leak}-current. All the currents with their notation are displayed in table 2.2, and the diagram in figure 2.6 outline how the cell model is set up.

Mathemathically, several of the ten Tusscher & Panfilov currents use Hodgkin-Huxley formulations. A generalized formulation for an ion X would be

$$I_X = f_X(V, Y) \tag{2.27}$$

$$\frac{dy}{dt} = \alpha_y(1 - y) - \beta_y y \tag{2.28}$$

where f is some function of the membrane potential V (and usually the Nernst potential E_X), the conductance constant g_X and a set of gating variables Y. A

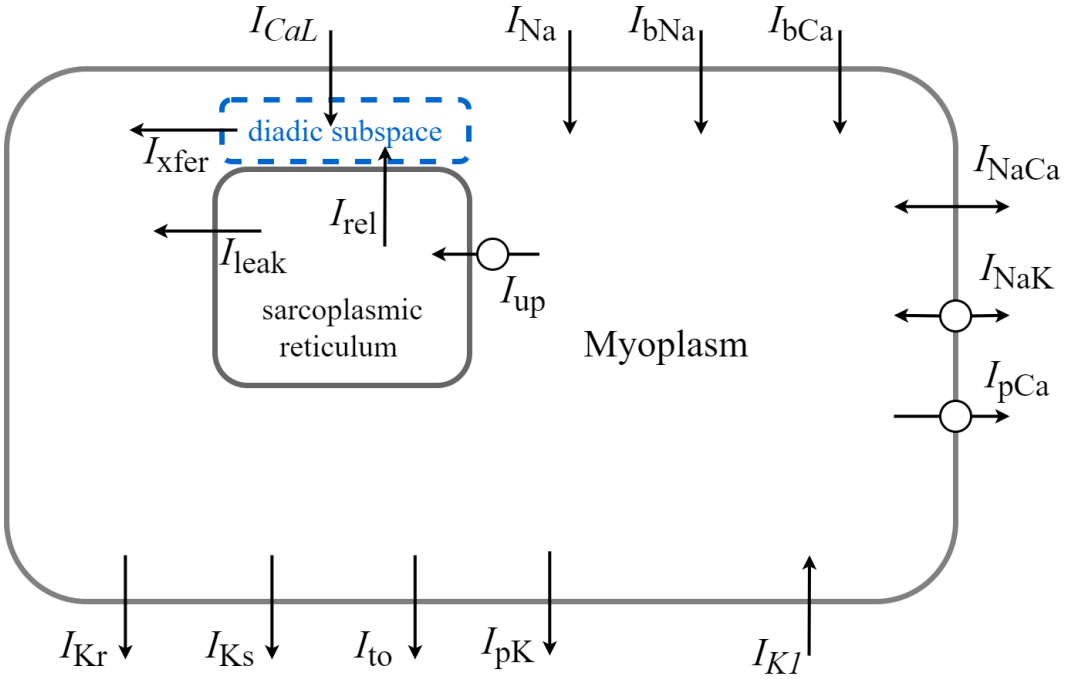


Figure 2.6: A diagram of the cell model desbribed in [Ten Tusscher and Panfilov, 2006], displaying the transmembrane currents and the intracellular calcium currents between the myoplasm and the SR. Pumps are marked with a circle over the arrow. For a description of each current, see table 2.2.

gating variabel $y \in Y$ satisfies a first order ODEs of some voltage dependant functions α_y, β_y, which are found by fitting to experimental data. The currents that use these formulations in pure form are I_{Na}, I_{to}, I_{Ks}, I_{bNa} and I_{bCa}.

Other currents are modelled with expressions that include a set of concentrations Z of the different ions in the myoplasm, the extracellular space, and the SR. A generalized formulation for an ion X for these currents would be

$$I_X = f_X(V, Z) \tag{2.29}$$

$$\frac{dz}{dt} = f_z(V, I_{\mathrm{ion}}, z) \tag{2.30}$$

Where, f_X is some function of the membrane potential V and a set of the relevant ion concentrations Z. An ion concentration $z \in Z$ satisfies some ODEs described by some function f_z that generally is non-linear, and dependant on the potential V the ion currents I_{ion}.

The total ion current in and out of the cell is given as

$$I_{\text{ion}} = I_{\text{Na}} + I_{\text{K1}} + I_{\text{to}} + I_{\text{Kr}} + I_{\text{Ks}} + I_{\text{CaL}} + I_{\text{NaCa}} + I_{\text{NaK}} + I_{\text{pCa}} + I_{\text{pK}} + I_{\text{bCa}} + I_{\text{bNa}} \quad (2.31)$$

In total, the ion current is dependant on 19 variables that all satisfy their own ODEs. These variables include the transmembrane potential, the gating variables and the ion concentrations, and are presented in table 3.3. The variables that describe the state of the cell and the ion current, and are often grouped together in the term *state variables*. Many of the state variables ODEs contains multiple other sub-variables, that again are functions of V. As the implementation of the TT2 model is well integrated in both softwares used in this thesis, a thorough examination of the large set of equations and functions that make up the model is not the main focus, and we will therefore not go through all of it here.

The important thing to take notice of, is the general formulations of the currents as they are important when determining the choice of numerical solution method. We refer to the appendix of [Tusscher et al., 2004] for the formulation of I_{Na}, I_{to}, I_{Kr}, I_{K1}, I_{NaCa}, I_{NaK}, I_{pCa}, I_{pK}, I_{bNa} and I_{bCa}, and to [Ten Tusscher and Panfilov, 2006] for I_{CaL}, I_{Ks}, I_{leak}, I_{up}, I_{rel} and I_{xfer}.

2.2.4 The Bidomain and Monodomain Model

So far, we have covered how to mathematically model the transmembrane potential and the currents going through the membrane of a single cell during an action potential. What remains however, is to mathematically describe the propagation of the potential along the cell membrane, and from cell to cell across the cardiac tissue.

First introduced in 1978, the *bidomain* model divides the conduction along the membrane into two domains with intra- and extracellular potential (V_i,V_e) and current (I_i, I_e) [Tung, 1978]. It makes the assumption that not only the extracellular, but also the intracellular space, is continuous across multiple cells. This assumption is made because the *gap junctions* where the cardiac cells meet have been shown to be of very low resistance, so that ions can pass from cell to cell unhindered to such a degree that an approximation using a continuous intracellular space with an average conductance is applied [Keener and Sneyd, 1998e]. This way of viewing cardiac tissue introduced the ability to create computationally achievable and clinically relevant simulations, and the bidomain model is in use to this day [Henriquez and Ying, 2021].

The relationship between current and potential follows Ohms law, and the currents in each domain are given by

$$I_i = -M_i \nabla V_i \qquad I_e = -M_e \nabla V_e \tag{2.32}$$

where M_i and M_e are conductivity tensors. Cardiac tissue is anisotropic, and will have different conductive properties in the different directions, so that for example

$$M_i = \begin{pmatrix} \sigma_{i,l} & 0 & 0 \\ 0 & \sigma_{i,t} & 0 \\ 0 & 0 & \sigma_{i,n} \end{pmatrix} \tag{2.33}$$

where σ_l is the longitudinal conduction, σ_t is the transversal conduction, and σ_n is the conduction normal to plane. The directions l, t and n are relative to the fiber orientation of the heart muscle cells in the local area in which we are calculating the conduction.

The the transmembrane potential at any point along the membrane is

$$V = V_i - V_e \tag{2.34}$$

If no external stimuli current is applied, the total current is conserved.

$$\nabla \cdot I_{\text{tot}} = \nabla \cdot (I_i + I_e) = 0 \tag{2.35}$$

$$\nabla \cdot (M_i \nabla V_i + M_e \nabla V_e) = 0 \tag{2.36}$$

Any current that enters or leaves the intracellular or extracellular space, must do so through the membrane, and is the transmembrane current I_τ

$$I_\tau = \nabla \cdot (M_i \nabla V_i) = -\nabla \cdot (M_e \nabla V_e) \tag{2.37}$$

Already having an expression for the transmembrane current in (2.2), and combining it with (2.37), we get

$$I_\tau = \chi \left(C_m \frac{dV}{dt} + I_{\text{ion}} \right) = \nabla \cdot (M_i \nabla V_i) \tag{2.38}$$

Where V is the transmembrane potential, C_m is the membrane capacitance, M_i and V_i is the conductance tensor and potential for the intracellular domain, and χ

the surface-to-volume ratio, which we need for (2.2) to be converted into units of current per volume.

The two equations (2.36) and (2.38) make up the bidomain model. A popular reduction of the bidomain model, is the *monodomain* model. It is based on the assumption that there is a constant scalar (λ) relation between the intracellular and extracellular conductions, so that, $M_e = \lambda M_i$. If that is the case, then the the intracellular and extracellular domain can be combined, using a harmonic mean of the two conduction tensors,

$$M_* = M_i(M_i + M_e)^{-1}M_e \tag{2.39}$$

We refer to [Keener and Sneyd, 1998e] where the detailed steps to derive the monodomain model are explained, but the resulting expression is

$$\chi\left(C_m\frac{dV}{dt} + I_{\text{ion}}\right) = \boldsymbol{\nabla}\cdot(M_*\boldsymbol{\nabla}V) \tag{2.40}$$

Having now introduced the monodomain model, which is the choice of cardiac model used by [Niederer et al., 2011], we have covered all the relevant equations needed to build the case we are going to implement and use to benchmark the two solvers.

2.2.5 A Summary of the Relevant Equations

At any point of a three dimensional slice of ventricular cardiac tissue, the membrane potential V is given by

$$\chi\left(C_m\frac{dV}{dt} + I_{\text{ion}}(V,Y,Z)\right) = \boldsymbol{\nabla}\cdot(M_*\boldsymbol{\nabla}V) \tag{2.41}$$

where C_m is the membrane capacitance, χ is the surface to volume ratio, V is the membrane potential, I_{ion} is the ion current, and M_* is the harmonic mean of the intracellular and extracellular conductances.

I_{ion} is a function describing the ion current in and out of the cell, and is the sum of all the underlying ion currents included in the cell model we choose.

$$I_{\text{ion}} = \sum_i I_i(V, Y, Z) \tag{2.42}$$

Where Y is a set of gating variables $Y_1, \ldots, Y_N$ that satisfy ordinary differential equations on the form

$$\frac{dY_i}{dt} = \alpha_i(1 - Y_i) - \beta_i Y_i \quad , \tag{2.43}$$

and Z is a set of ion concentrations $Z_1, \ldots, Z_M$ that satisfy another set of ordinary differential equations

$$\frac{dZ_i}{dt} = f_i(V, I_{\text{ion}}, Z_i) \tag{2.44}$$

where V again is the potential and I_{ion} are the ion currents in and out of the cell. It is also common to denote the ion current as $I_{\text{ion}}(V, S)$ where S are the so called *state variables*, which encompasses both the gating variables and the concentrations.

In this case we use the ten Tusscher & Panfilov 2006 model, in which

$$I_{\text{ion}} = I_{\text{Na}} + I_{\text{K1}} + I_{\text{to}} + I_{\text{Kr}} + I_{\text{Ks}} + I_{\text{CaL}} + I_{\text{NaCa}} + I_{\text{NaK}} + I_{\text{pCa}} + I_{\text{pK}} + I_{\text{bCa}} + I_{\text{bNa}} \tag{2.45}$$

To summarize, we are dealing with a set of coupled non-linear ODEs and PDEs that must be discretized in both space and time, and a potential with rapid changes and steep gradients. Such a system is a challenge to solve numerically in an efficient way.

2.3 Numerical Methods

In this section we will introduce the relevant numerical methods used to solve the set of equations above. We will discuss the computational cost, accuracy and limitations of some approaches, and sketch out the algorithm we are going to implement to perform our simulations. A lot of the theory covered in this section is from *Computing the electrical activity in the heart* [Sundnes et al., 2007b].

By moving around terms in (2.41), we have

$$\frac{dV}{dt} = \boldsymbol{\nabla} \cdot (D\boldsymbol{\nabla} V) - \frac{1}{C_m} I_{\text{ion}}(V, Y, Z) \tag{2.46}$$

where D is $M_*/(\chi C_m)$

2.3.1 Operator Splitting

When dealing with a coupled system of ODEs and PDEs *operator splitting* is an attractive technique, as it deconstructs a complex set of equations into smaller, easier solvable parts. It divides the problem into two operators, one associated with the diffusive and one with the non-diffusive part of the equation, and solves them separately. If the previous step is t_n, the different parts of the equations are solved at steps between t_n and $t_n + \Delta t$. The point at which equations are solved are given by the value θ. Strang-splitting is $\theta = 0.5$ and gives 2nd-order accuracy [Sundnes et al., 2007a].

An algorithm for operator splitting on the monodomain model is proposed by [Qu and Garfinkel, 1999], and consists of the following steps:

(I) Solve the set of ion current equations

$$\frac{dV}{dt} = -\frac{1}{C_m} I_{\text{ion}}(V, Y, Z) \tag{2.47}$$

$$\frac{dY_i}{dt} = \alpha_i(1 - Y_i) - \beta_i Y_i \qquad i = 1, \ldots, N \tag{2.48}$$

$$\frac{dZ_i}{dt} = f_i(V, I_{\text{ion}}, Z_i) \qquad i = 1, \ldots, M \tag{2.49}$$

$$V(t_n) = V^n \quad Y(t_n) = Y^n \quad Z(t_n) = Z^n$$

at $t = t_n + \theta\Delta t$, with V, Y and Z from the previous iteration as initial conditions. The results are denoted as V_θ^n, Y_θ^n and Z_θ^n

(II) Solve the PDE

$$\frac{dV}{dt} = \boldsymbol{\nabla} \cdot (D\boldsymbol{\nabla} V) \tag{2.50}$$

$$V(t_n) = V_\theta^n$$

at $t = t_n + \Delta t$ using the result from step (I) as initial condition, and denoting the result as V_θ^{n+1}.

(III) Finally, solve the last half step of the the ion current equations, repeating step (I), but integrating from $t_n + \theta\Delta t \longrightarrow t_n + \Delta t$ using

$$V(t_n + \theta\Delta t) = V_\theta^{n+1} \quad Y(t_n + \theta\Delta t) = Y_\theta^n \quad Z(t_n + \theta\Delta t) = Z_\theta^n$$

as initial conditions, and obtaining the final solution of the iteration V^{n+1}, Y^{n+1} and Z^{n+1}.

Generally, we use $\theta = 0.5$ so we are solving the ODEs of the ion currents in the cells every half time step, and the PDE of the diffusion along the membrane every whole step. A big advantage of this splitting, is that we can apply different numerical solution methods to the different steps, choosing something that is optimal and efficient for the time discretized ODE and the space discretized PDE. As using $\theta = 0.5$ gives 2nd order accuracy, this is the natural level of accuracy to aim for when solving the individual parts as well. The result will ultimately be bound by the accuracy of the operator splitting, so solving with higher order accuracy for the individual equations is redundant.

2.3.2 Solving the PDE

When solving a partial differentiall equation, we typically use either finite difference or finite element methods. In both carp and cbcbeat we use the finite element method (FEM) with tetrahedral grid elements for spatial discretization. An advantage of FEM is that it handles the irregular geometries of the heart and body more conveniently than the finite difference method. The time discretization of (2.50) is

$$\frac{V^{n+1} - V^n}{\Delta t} = \theta\big(\boldsymbol{\nabla} \cdot (D\boldsymbol{\nabla}V^{n+1})\big) + (1-\theta)(\boldsymbol{\nabla} \cdot (D\boldsymbol{\nabla}V^n)) \tag{2.51}$$

where V^{n+1} is the next step, and V^n is the previous step. Using $\theta = 0.5$ we have the Crank-Nicolson scheme.

Creating a Linear System

In short, using the finite element method, we introduce a function space V where $V^{n+1} \in V$. We multiply with an arbitrary test function $\psi \in V$, and integrate over

the domain H. Applying Greens lemma and a zero-flux boundary condition, we are left with the problem of finding $V^{n+1} \in V$ so that

$$\int_H V^{n+1}\psi dx + \theta\Delta t \int_H D\boldsymbol{\nabla}V^{n+1} \cdot \boldsymbol{\nabla}\psi dx = \int_H V^n\psi dx - (1-\theta)\Delta t \int_H D\boldsymbol{\nabla}V^n \cdot \boldsymbol{\nabla}\psi dx \tag{2.52}$$

for all $\psi \in V$. For the detailed steps, see [Sundnes et al., 2007a], page 77.

Then, using a set of basis functions ϕ_j spanning V_h and a set of scalars c_i, where $j = 1, \ldots, n$, and $V_h \subset V$, we can approximate V^{n+1} as a linear combination of these basis functions

$$V^{n+1} = \sum_{j=1}^{n} c_j\phi_j \tag{2.53}$$

which allows us to write (2.52) as a system of linear equations

$$\boldsymbol{A}V = b \tag{2.54}$$

where

$$A_{i,j} = \int_H \phi_j\phi_i dx + \theta\Delta t \int_H D\boldsymbol{\nabla}\phi_j \cdot \boldsymbol{\nabla}\phi_i dx \tag{2.55}$$

$$b_i = \int_H V^n\phi_i dx - (1-\theta)\Delta t \int_H D\boldsymbol{\nabla}V^n \cdot \boldsymbol{\nabla}\phi_i dx \tag{2.56}$$

The two parts of $\boldsymbol{A}$ are often called the *stiffness matrix* $\boldsymbol{K}$, where

$$K_{i,j} = \int_H D\boldsymbol{\nabla}\phi_j \cdot \boldsymbol{\nabla}\phi_i dx \tag{2.57}$$

and the mass matrix $\boldsymbol{M}$, where

$$M_{i,j} = \int_H \phi_j\phi_i dx. \tag{2.58}$$

With (2.54), the PDE part of the monodomain equation is reduced to solving a system of linear equations.

Conjugate Gradient

There are many possible choices of iterative methods one can apply when aiming to solve a linear system. A good choice in our case, is the *conjugate gradient* (CG)

method, which if well conditioned, can reach an approximate solution within a small number of iterations [Faul, 2018].

We define two inner products

$$(u, v) = \frac{1}{N} \sum_{i=1}^{N} u_i v_i \qquad (u, v)_A = (Au, v) \tag{2.59}$$

Their respective norms are given by

$$\|u\| = (u, u)^{1/2} \qquad \|u\|_A = (\boldsymbol{A}u, u)^{1/2}, \tag{2.60}$$

where $(u, v)_A$ is the so called A-norm.

We aim to solve (2.54), which is

$$\boldsymbol{A}V = b \tag{2.61}$$

Let W be a subspace of $\mathbb{R}^N$. If w is the best approximation of V in the A-norm, meaning

$$\|V - w\| \leq \|V - v\| \qquad \forall v \in W \tag{2.62}$$

then the error is orthogonal to the subspace W

$$(V - w, v)_A = 0 \qquad \forall v \in W \tag{2.63}$$

Let

$$W_k = \text{span}\{p_0, \ldots, p_{k-1}\} \tag{2.64}$$

Where $p_0, \ldots, p_{k-1}$ are k vectors called *search vectors*, that are orthogonal in the A-norm.

Assume that $w_k \in W_k$ is the current best approximation to V. The residual is then defined as

$$r_k = b - \boldsymbol{A}w_k \tag{2.65}$$

which, as defined by the equations above, is A-orthogonal to all the vectors $p_0, \ldots, p_{k+1}$.

The conjugate gradient method uses the residual and the Gram-Schmidt algorithm to calculate new orthogonal basis vectors, and by that new approximations w_{k+1}. This is repeated until the error is sufficiently small, governed by some tolerance $0 < \varepsilon < 1$.

The *condition number* K of the matrix $\boldsymbol{A}$ is governed by the largest and smallest eigenvalues of $\boldsymbol{A}$, as

$$K = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{2.66}$$

The value is of great importance because if K is small (close to 1), the problem is *well conditioned*, and the solution can converge very fast. If K is large on the other hand, the converge can become very slow. K is inherent to the problem we aim to solve, and is typically $\mathcal{O}(h^{-2})$, h being the grid spacing.

In order to increase the convergence rate, *preconditioning* is often applied. The idea is to multiply the problem (2.54) with a matrix B, called a preconditioner, and instead solve the equation

$$\boldsymbol{BA}V = \boldsymbol{B}b \tag{2.67}$$

where $\boldsymbol{B}$ is some approximation of $\boldsymbol{A^{-1}}$. The choice of $\boldsymbol{B}$ can vary, but the goal is to reduce the condition number so that $K(BA) < K(A)$, and by that decrease the number of necessary iterations before an acceptable approximation of V is achieved. The *block Jacobi* preconditioner, which is the one we are going to use in our numerical experiments, is $\boldsymbol{B} = \boldsymbol{D}^{-1}$, where $\boldsymbol{D}$ is a diagonal matrix with the block matrices on the diagonal of $\boldsymbol{A}$. The details of the preconditioned conjugate gradient method is given in algorithm 1.

Both the solvers, as we shall describe in more detail in the next chapter, utilize PETSc [Abhyankar et al., 2018] for the conjugate gradient method and its preconditioner.

Algorithm 1: The preconditioned conjugate gradient method, as given in [Sundnes et al., 2007c], to solve (2.50), via (2.67)

Let V_n and b_n be the membrane potential and the right hand side of the equation at a time t_n. Let w_0 be the initial approximation of V_{n+1}, and let V_{n+1}^k denote the k-th approximation. Let ε be some tolerance $0 < \varepsilon < 1$.

$V_{n+1}^0 = w_0$
$r = b_n - AV_{n+1}^0$
s = Br
$p = s$
$\rho_0 = (s, r)$
$k = 0$
while $\rho_k/\rho_0 > \varepsilon$ **do**
    $z = Ap$
    $t = Bz$
    $\alpha = \rho_k/(p, z)$
    $V_{n+1}^{k+1} = V_{n+1}^k + \alpha p$
    $r = r - \alpha z$
    $s = s - \alpha t$
    $\rho_{k+1} = (s, r)$
    $\beta = \rho_{k+1}/\rho_k$
    $p = s + \beta p$
    $k = k + 1$
end

2.3.3 Solving the ODEs

The ODE part of the equation, is

$$\frac{dV}{dt} = -\frac{1}{C_m} I_{\text{ion}}(V, Y, Z) \tag{2.68}$$

where Y and Z are sets of in total 18 state variables that, as described in (2.48) and (2.49), satisfy their own differential equations.

The general approach when solving such equations numerically, is to make an approximation of the derivative by discretization to some order of Δt. Typical strategies for higher order accuracy could be the Runge Kutta methods, while a more simple approach is the well known forward Euler (FE) method.

Generally, the ODEs describing the gating variables Y and the concentrations Z, are non-linear, that is, they take the form

$$\frac{dy}{dt} = f(y, S) \qquad y(V_0) = y_0 \tag{2.69}$$

Where f is some non-linear function dependant on y and potentially a set S of other variables, i.e. ionic concentrations.

The forward Euler discretization of this equation is

$$\frac{u_{n+1} - u_n}{\Delta t} = f(u_n, S_n) \tag{2.70}$$

$$u_{n+1} = u_n + \Delta t f(u_n, S_n) \tag{2.71}$$

Using FE is a tempting option, because it with its simplicity is very computationally efficient. A challenge in cardiac electrophysiolgy is however, that the equations describing the ionic currents and cellular reactions make the problem very *stiff*. Stiffness is related to the eigenvalues of the Jacobian of f, and indicates whether a problem is prone to numerical instability and in need of very small time steps. A result of this is that higher order accuracy and more computationally heavy methods are needed, unless we can reduce the stiffness of the ODE system.

The Rush Larsen Scheme

A popular scheme to increase the efficiency and stability of the ODE solver, is the Rush-Larsen (RL) scheme [Rush and Larsen, 1978], which uses multiple techniques to reduce the computational load.

Firstly they propose an adaptive time step. As discussed earlier, and shown in figure 2.4, the action potential has areas of very rapid change in the membrane potential where the problem is very stiff and in which a small Δt is needed. However, in the less action-filled parts of the action potential, one can get away with a larger time step and still achieve acceptable accuracy. The time step is chosen by monitoring dV/dt, increasing the time step for periods of small change, and thus reducing the total number of iterations necessary.

Secondly, and most importantly, RL is a *partitioned method* that introduces a way divide the state variable ODEs into two categories. The reason behind this approach is the observation that only a fraction of the ODEs in a model actually contribute to the high stiffness of the total set. As an example, only 1 (the m gate) out of the 19 variables in the original ten Tusscher and Panfilov 2004 model have a very stiff ODE [Marsh et al., 2012]. By making this distinction, the non-stiff ODEs can be solved by the inexpensive forward Euler method, which can drastically reduce the computational load.

Noticing that for many models, the gating variables were the main contributors to the stiffness of the problem, Rush and Larsen proposed the following:

The the gating variables y all generally satisfy, as we have seen, ODEs on the form

$$\frac{dy}{dt} = \alpha_y(1 - y) - \beta_y y \tag{2.72}$$

If α_y and β_y were to be constants, and not functions, we could rewrite (2.72) as

$$\frac{dy}{dt} = \alpha_y - y(\alpha_y + \beta_y) \tag{2.73}$$

which again can be rewritten as

$$\frac{dy}{dt} = \frac{y_\infty - y}{\tau_y} \tag{2.74}$$

$$y_\infty = \frac{\alpha_y}{\alpha_y + \beta_y} \qquad \tau_y = \frac{1}{\alpha_y + \beta_y}$$

For (2.74) to hold up, α_y and β_y which are functions of the transmembrane potential V, must be assumed to be constant over dt. If that is the case, then all the gating variables taking the form of (2.72) can be treated as a linear ODE, and be solved by an exponential integration step

$$y_n = y_\infty - (y_{n-1} - y_\infty)e^{-(\Delta t/\tau_y)} \tag{2.75}$$

with y_∞ and τ_y as in 2.74 [Rush and Larsen, 1978].

And with that, we have the central idea of the Rush-Larsen scheme. By making the approximation of α_y and β_y (and by that V) to be constant over the time step Δt, the stiff gating variable ODEs are rewritten to be solved via an exponential integrator, while the rest of the state variable ODEs are solved with the tried and trusted forward Euler method. See algorithm 2

Algorithm 2: The Rush-Larsen scheme at a given time t_n as introduced by [Rush and Larsen, 1978], to solve (2.47) - (2.49).

Let $I_{\text{ion}}(V, S)$ be a function of the transmembrane potential V and a set of state variables S. Let $Y \subset S$ be the set of gating variables, and $Z \subset S$ be the remaining state variables. Let f_s be a functions describing the behaviour of a state variable $s \in S$. Let n denote the variables at a time t_n.

for ***each*** $s_{n-1} \in S_{n-1}$ **do**
  if $s_{n-1} \in Y_{n-1}$ **then**
    $_n = \alpha_s(V_n)$
    $\beta_{sn} = \beta_s(V_n)$
    $s_\infty = \alpha_{sn}/(\alpha_{sn} + \beta_{sn})$
    $\tau_s = 1/(\alpha_{sn} + \beta_{sn})$
    $s_n = s_\infty - (s_{n-1} - s_\infty)e^{-(\Delta t/\tau_s)}$
  end
  if $s_{n-1} \in Z_{n-1}$ **then**
    $s_n = s_{n-1} + \Delta t f_s(s_{n-1}, V_n)$
  end
end
$I_n = I_{\text{ion}}(V_n, S_n)$
$V_{n+1} = V_n + \Delta t I_n$

2.3.4 A Summary of the Solution Steps

Having now introduced all the mathematics and the necessary numerical methods needed to solve them, we can present an overview of the solution steps that are to be implemented in the two solver softwares.

The main parts of a step $t_n \rightarrow t_n + \Delta t$ when solving the monodomain model (2.41) given the methods introduced in this chapter is:

1. Using Strang splitting $\theta = 0.5$ to solve the ODE part of the equation (2.47) - (2.49) at $t = t_n + \theta\Delta t$ at all points in the domain. This is done with the Rush-Larsen scheme, using an exponential integrator for the gating variables Y and FE for the remaining state variables Z as shown in algorithm 2.

2. Solve PDE (2.50) at $t = t_n + \Delta t$ with the conjugate gradient method preconditioned with the block Jacobi method as shown in algorithm 1, using V resulting from step 1) as initial condition.

3. Repeat step 1) for the last half of the θ-split, using V from 2) and Y and Z from 1) as initial conditions to merge the results and obtain the final solution.

2.3.5 Computational Cost

Before we cover how the numerical methods above are implemented in the two solvers, it is interesting to take a qualitative look at the different operations included and how much we can typically expect them to contribute to the total computational cost.

There are a few main parts of the Rush-Larsen scheme we should keep an eye on regarding efficiency. Firstly, the scheme has to loop over each node in the grid/domain, which should be done in a practical way. Secondly, there are a lot of values that need to be calculated for the right hand side of the integrations for each time step. The in total 18 state variables (excluding V) all have functions that often are dependant on subvariables that again are dependant on V. This can lead to a lot of expensive function calls on each time step if not implemented efficiently. Finally the integration steps, which is either FE or the exponential integrator, depending on the state variable. When all the variables for the right hand side is calculated, this should not be a particularly expensive operation.

For the PDE, the conjugate gradient method generally consists of a series of repeated linear operations that should be solved in an optimal way to ensure efficiency. The preconditioner $\boldsymbol{B}$ and the matrix $\boldsymbol{A}$ are mainly computed just once when the solver is initialized. The right hand side of the equation b, however, needs to be computed and assembled for each new time step. This is an area in which a lot of computational time can be added, if the matrix assembly is not optimized.

In the next chapter, we are going to take a look at cbcbeat and openCARP and how the numerical methods we have discussed are implemented in each of them.

Chapter 3

Benchmarking openCarp and CBCbeat

In this chapter, we cover how the benchmark was created in the two solvers. First, we present the benchmark case, as defined in the [Niederer et al., 2011] article, with all the models, conditions, geometry and parameters used. As well as how verification of implementations are done through measuring activation times at certain points on a tissue slice.

This chapter also introduces the software codes for the two solvers *cbcbeat* [Marie Rognes et al., 2017] and *openCARP* [Plank et al., 2021b], in which we are going to set up and time our benchmark. It covers the the origin and background of the solvers, as well as an overview of their architecture and and the elements most relevant to the case we are implementing in this thesis. We will also bring attention to some similarities and differences between them that are relevant to their efficiency.

As every detail about the solvers cannot possibly be covered in this chapter, we refer to the openly available documentation. It should be noted, that there is a significant difference in the documentation available for the two solvers. OpenCARP has an official website [Plank et al., 2021a] with documentation, user manuals, examples and more, as well as open repositories for the different components. Cbcbeat has a fairly informative read-the-docs page [ME Rognes et al., 2017], but it does not currently have an official open repository, though various repositories of different earlier versions exist. The FEniCS project on which cbcbeat is based, however, is extensively documented, including a tutorial book [Langtangen and Anders Logg, 2016].

Lastly, in this chapter we explain how the benchmark has been implemented in cbcbeat and openCARP, and how our codes are verified. We go through how the two solvers were timed, and considerations that were made along the way.

3.1 An Overview of the Benchmark Case

The Niederer benchmark is a great case for verifying cardiac electrophysiology simulation software, as its provides a standardized problem with a converging numerical solution. Furthermore, it is not too computationally heavy, and contains no bias for a specific numerical scheme.

Mathematical Model

The Niederer benchmark uses the monodomain model for the electrical activation of the tissue, and the epicardial ten Tusscher & Panfilov 2006 (TT2) model for the ion currents and intracellular calcium dynamics. All models are presented in section 2.2, but for convenience, we repeat the equation here.

The monodomain model equation is

$$\chi\left(C_m\frac{dV}{dt} + I_{\text{ion}}(V, Y, Z)\right) = \boldsymbol{\nabla}\cdot(M_*\boldsymbol{\nabla}V) \tag{3.1}$$

where M_* is a matrix containing the directional conductivities σ, and the ion current is given by the TT2 model as described in section 2.2.3.

The constants and variables with their symbol and units are given in table 3.1, and the benchmark specific values of the constants in table 3.2

Constants and Variables of the Monodomain Model		
symbol	physical meaning	unit
χ	surface-to-volume ratio	mm^{-1}
C_m	membrane capacitance	$\mu\mathrm{F}/\mathrm{mm}^2$
V	membrane potential	mV
I_{ion}	transmembrane current	$\mu\mathrm{A}/\mathrm{mm}^2$
σ_l	longitudinal conductivity	mS/mm
σ_t	transverse conductivity	mS/mm
t	time	ms
x, y, z	distance	mm

Table 3.1: The constants and variables in the monodomain model, with their units. Some implementations may use cm or μm in stead of mm.

Constant Values in the Benchmark		
constant	value	unit
χ	140	mm^{-1}
C_m	0.01	$\mu\mathrm{F}/\mathrm{mm}^2$
σ_l (intra)	0.17	mS/mm
σ_l (extra)	0.62	mS/mm
σ_t (intra)	0.019	mS/mm
σ_t (extra)	0.24	mS/mm

Table 3.2: The values for the monodomain model constants used in the Nieder benchmark. The conductivities are calculated as the harmonic mean of their intra- and extracellular values $\sigma_l = \sigma_{l,i}\sigma_{l,e}/(\sigma_{l,i} + \sigma_{l,e})$.

Geometry

The tissue model used in the benchmark, is a cuboid of dimensions $3 \times 7 \times 20$ mm. The conductivity is represented with an anisotropic conductivity tensor with its principle axes aligned with cuboid. The Niederer benchmark uses a bidirectional conductivity, with a longitudinal conductivity aligned with the long (20 mm) axis, and a transverse conductivity along the two other axes.

When stimulation is applied, it is applied in a $1.5 \times 1.5 \times 1.5$ mm cube located in one of the corners of the tissue slice, as shown in figure 3.1a.

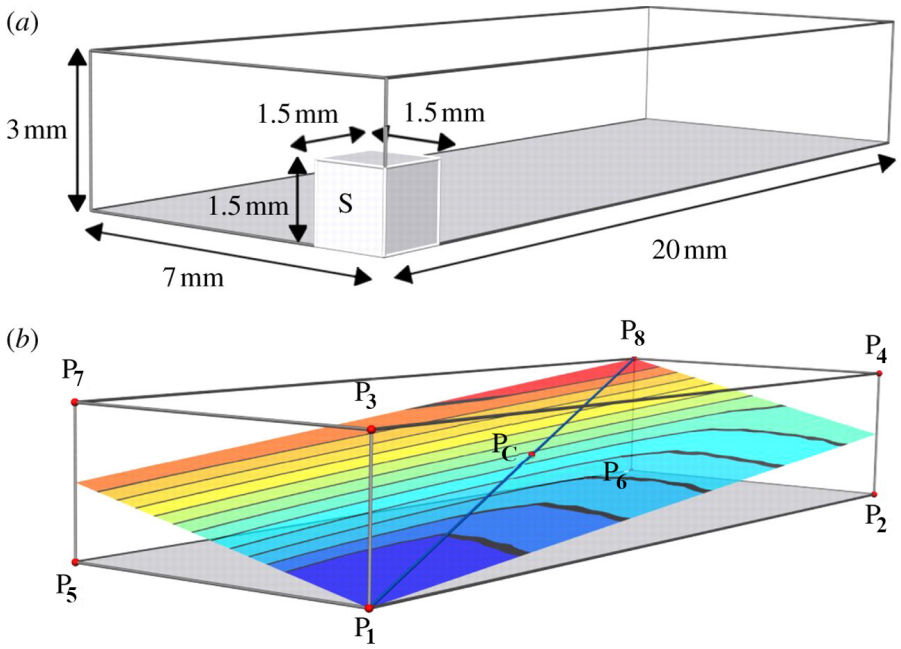


Figure 3.1: a) The dimensions of the tissue cuboid, and the placement and dimensions of the stimuli area. b) The location of the nine points where activation time is measured, as well as the P1-P8 line with an illustration of the potential propagation in the tissue. Figure adapted from [Niederer et al., 2011] under a CC-BY license [CreativeCommons, 2021a] as given by The Royal Society's open access article guidelines [TheRoyalSociety, 2021].

Intial- and boundry conditions

The benchmark assumes all boundaries to have zero-flux boundary conditions. As for the initial values of variables and states, they are given in table 3.3. For the equations in which all the gating variables and concentrations are used, see [Tusscher et al., 2004] and [Ten Tusscher and Panfilov, 2006].

symbol	description	initial value
V	membrane potential	-85.23 mV
x_{r1}	rapid time-dependent potassium current gate	0.00621
x_{r2}	rapid time-dependent potassium current gate	0.4712
x_{rs}	slow time-dependent potassium current gate	0.0095
m	fast sodium current gate	0.00172
h	fast sodium current gate	0.7444
j	fast sodium current gate	0.7045
d	L-type Ca current gate	$3.373 \cdot 10^{-5}$
f	L-type Ca current gate	0.7888
$f2$	L-type Ca current gate	0.9755
$fCass$	L-type Ca current gate	0.9953
s	transient outward current gate	0.999998
r	transient outward current gate	$2.42 \cdot 10^{-8}$
R_{prime}	ryanodine receptor	0.9073
Ca_i	intracellular calcium concentration	0.000126 mM
Ca_{SR}	sarcoplasmic reticulum calcium concentration	3.64 mM
Ca_{ss}	subspace calcium concentration	0.00036 mM
Na_i	intracellular sodium concentration	8.604 mM
K_i	intracellular potassium concentration	136.89 mM

Table 3.3: The initial values of the state variables in the TT2 model, which include the potential, as well as the gating variables and initial ion concentrations.

Protocols and Measurements

The benchmark simulation itself consists of the following:

At $t = 0$ a stimulus current of strength $50000\mu\text{A/cm}^3$ is applied to the area S as shown in figure 3.1a) for 2 ms. The potential change should propagate through the tissue. A point in the cuboid is considered activated when the potential at the given point rises above 0 mV. The time passed since the initial stimulus to the activation of a point is referred to as the *activation time* of that point.

Each corner of the cuboid, as well as the center, are given a name, with P1 being the corner of the initial stimulus, and P8 the opposite. The configuration of the points are shown in figure 3.1 b). The activation time of the nine points are recorded for all combinations of the spatial resolutions $\Delta x = [0.5, 0.2, 0.1]$ mm and temporal resolutions $\Delta t = [0.05, 0.01, 0.005]$ ms.

The activation times can verify the accuracy of the solvers for different resolutions as they converge towards a 'true' solution. There is no analytical solution to the model, and comparing to experimental observations is very challenging. The gold standard solution is therefore achieved by comparing all the results in a so called N-version benchmark. [Hatton and Roberts, 1994]. For the activation times of all the 11 software codes in the Niederer article, see the supplemental material of [Niederer et al., 2011].

Numerical Methods

The benchmark does not specify what numerical schemes are to be used to perform the simulation, and in the original article the participating solvers use different methods. In our case, we will use the same method in both cbcbeat and openCARP. We use operator splitting with Strang splitting ($\theta = 0.5$). The PDE is solved using the finite elements on a tetrahedral mesh and the conjugate gradient method with a block Jacobi preconditioner, while the ODEs are solved with the Rush-Larsen scheme.

In the upcoming sections, we will first give a general description of the two solvers' software, their architecture and the functionality most relevant to our work. Then, in section 3.5, we will give a more detailed description of how we have implemented the specific Niederer benchmark in both cbcbeat and openCARP.

3.2 cbcbeat

Developed at Simula Research Laboratory in Oslo, Norway, cbcbeat is a collection of python-based software specifically targeting computational cardiac electrophysiology problems. It utilizes the FEniCS project [A. Logg et al., 2012] [Alnæs et al., 2015] as its core.

FEniCS is developed with the aim of creating efficient and flexible software for solving PDEs using finite element methods. It consist of a high-performance C++ backend that can be accessed from both C++ and python problem solving environments. In the backend of FEniCS we find DOLFIN [Anders Logg and Wells, 2010], which handles data structures such as function spaces, functions and meshes, as well as compute algorithms like finite element assembly. Additionally, FIAT is a finite element backend generating finite element functions, while mshr provides mesh generation capabilities. FEniCS is compatible with PETSc [Abhyankar et al., 2018], a state of the art library for complex differential equations projects which allows for careful costumization and a plethora of numerical method options.

Cbcbeat itself is made up of a set of python modules with classes for handling the different steps in solving cardiac electrophysiology equations. When a numerical experiment is set up, cbcbeat let us choose a variety of parameters that determine how the different modules will behave. As the monodomain model is the focus of this thesis, we will be introducing the classes relevant to solving our specific case. Figure 3.2 presents an overview of the architecture of the cbcbeat solver.

Relevant Cbcbeat Classes and DOLFIN Functions

The Cell Model

Cell models such as ten Tusscher & Panfilov are implemented as subclasses of a class `CardiacCellModel` in the *cardiacmodels* module. The classes contain a dictionary with all the parameters of the model, and comes with a set of default initial conditions that we can override if we so wish. The huge set of functions that make up I_{ion}, is handled in the cell model class methods `I()` and `F()`, where the `I(v,s,time)` returns the total ion current, while `F(v,s,time)` returns a vector containing the individual expressions for the 18 state variables. An important detail is that both methods return the result as UFL expressions. The Unified Form Language is a part of the FEniCS package and is a domain specific language

used to declare functions on finite element discretizations.

The whole cardiac system is then created as an instance of the class `CardiacModel`, using the chosen cell model, as well as parameters such as the mesh, conductivity tensor and stimulus protocol.

The ODE Solver

The *cellsolver* module contains the `CardiacODESolver` class which is a solver optimized for working with equations related to the ion currents. Using the ion current expressions from the cell model, it sets up the right hand side of the equation (2.47) as `rhs`, which is also a UFL expression. First the `Scheme` is initialized as an instance of some child class of the class `MultiStageScheme`, in our case RL1 (Rush-Larsen, see algorithm 2). Then, the solver is initiated as an instance of the `PointIntegralSolver` in DOLFIN with `rhs` and the `Scheme` given as the input parameters. The solve/step part of the solver simply calls the step function of `PointIntegralSolver`.

In the `step` function of DOLFIN's `PointIntegralSolver`, an outer loop goes over all the vertices in the grid. An inner loop goes through all the content of `rhs`, evaluates the expressions, and solves the integrals according to the Rush-Larsen scheme. Finally all the variables, the total solution, and the time is updated.

The PDE Solver

For the PDE, the *monodomainsolver* module contains the class `MonodomainSolver`. When initialized, if an iterative solver is chosen, it creates `solver` as an instance of the DOLFIN class `PETScKrylovSolver` with the algorithm and preconditioner of our choice as arguments. In our case, the algorithm is `'cg'` (conjugate gradient method), and the preconditioner is `'bjacobi'` (block Jacobi). It will also create the matrix $\boldsymbol{A}$ as `lhs_matrix`, and set it as the operator for `solver`.

The `step` function of `MonodomainSolver` will first check if the solver need to be updated (if there has been a change in the size of Δt), and do so if necessary. Then it will compute b for the given time step as `rhs`, and call on the `solve` function of `PETScKrylovSolver`, which uses PETSc to solve the linear system according to alogrithm 1. Details regarding the PETSc options can be specified using the *petsc4py* module. For the creation of the left hand side matrix $\boldsymbol{A}$, and the right

hand side vector b, the solver utilizes the `assemble` function which is imported from DOLFIN.

Operator Splitting & Complete Scheme

The *splittingsolver* module, which contains the `SplittingSolver` class use the modules mentioned above to solve the full equation. It is the top-level solver, in that it does not itself contain a numerical method to solve the equations. Rather, it takes the `CardiacModel`, and initializes a PDE and an ODE solver based on the input parameters it is given. It will proceed to call on the solvers according to the operator splitting algorithm (see section 2.3.1) and the chosen value of θ. Lastly, it merges the results to a final solution.

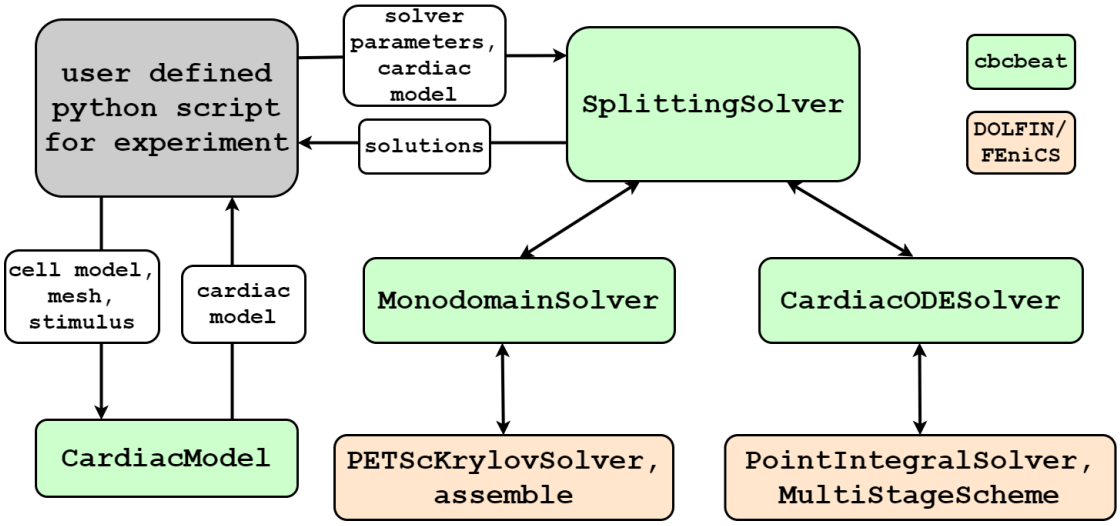


Figure 3.2: Chart showing the architecture of the cbcbeat solver when used for simulations of the monodomain model. An overview of the main classes, and how they call upon one another. Cbcbeat modules coded in python are shown in green, and DOLFIN modules coded in C++ are shown in orange.

In short, cbcbeat is a structure that allows us to easily choose between numerical schemes and solution methods that are relevant for cardiac electrophysiology simulations. The modules that create the path to the solution by splitting the problem, setting up the functions, choosing the solvers and iterate over time, are done in python. The more computationally heavy operations are left to the DOLFIN backend in C++ with the help of PETSc.

Cbcbeat also utilizes the DOLFIN class `Timer` well, which allows us to easily get valuable insight in not only the total computation time, but detailed information about the time spent on the various steps in the solution.

3.3 openCARP

Having been developed for cardiac electrophysiology research over the last 20 years the *openCARP* eco system is today comprised of a number of different components for handling meshes, simulations and visualization. It is an open source cardiac electrophysiology simulator for in-silico experiments, free to use for academic purposes [Plank et al., 2021b].

The openCARP simulator is coded in C++ and is the core of the software. When installed the compiled simulation application can be run directly from terminal, using a set of parameters outlining the specifics of a given experiment. A more sophisticated approach, which is recommended for larger projects, is to make use of the python framework *carputils*. That way, the user can determine all the details of an experiment in a python script, and use the carputils module to set up, format, and pass a list of of commands to the application using the python *subprocess* module. For handling meshes, openCARP makes use of the C++ coded command-line tool *meshtool*.

In addition to the above mentioned components, openCARP has various tools for visualization such as *meshalyzer* and *carputilsGUI*. They are, however, not used in the simulations for this thesis, and we will therfore turn our attention to the central parts of the openCARP simulator that are involved in solving the monodomain equations.

Main Classes and Components of the openCARP Simulator

The main components of the simulator is an ionic current solver used for solving the ODE and calculation of I_{ion}, and a parabolic solver which solves the PDE and calculates the propagation along the cell membrane. openCARP also has an elliptic solver, but it is not used in our simulations as we are only implementing the monodomain, and not the bidomain model.

Cell Models: the Limpet Library

The ionic current properties of the system is set up as an instance of the class `MULTI_ION_IF`, *multi-ion interface*, which is a part of the `limpet` library in the `ionics` module. The limpet library contains a large set of popular cell models, such as TT2, AP, and many others. The cell models come ready implemented with all parameters and and functions describing the variables. They also come with a model specific `compute` function, that partitions the variables according to the Rush-Larsen scheme and computes the associated integration step. For efficiency, openCARP uses lookup tables for a lot of the variabels and subvariables that are included in the calculation of I_{ion}, which would otherwise be calculated through expression evaluations or function calls.

The ODE Solver

The ODE describing the ion currents in and out of the cells are handled by the `ionics` module. When the ionics are initialized, a multi-ion interface (MIIF) is created with the appropriate cell model, as described above. The `compute_step` function in ionics, calls on the `compute_ionic_currents` function of the `MULTI_ION_IF` class. Here the operator splitting is done by and outer looping over sub-time-steps given by the splitting scheme. For each sub-time-step (2 steps for Strang splitting), the ODE is solved. The ionic currents are calculated in a loop which goes over each node in the mesh, for then to call on the `compute` function associated with the cell model that has been assigned to the MIIF, in our case TT2.

The PDE Solver

In openCARP, the conduction along the cell membrane is taken care of by the `electrics` module. The way the that the interaction between the ionics and electrics is set, up, it is the `electrics` functions that are called upon by the overlaying solver, which in turn calls on the `ionics` functions. As an example, when the `initialize` function of `electrics` is called, it will first call the initialize function of `ionics`. Then, it will set it self up, with the electric properties we have given in the input parameters, such as conductivity and stimulus. Likewise, the `compute_step` function of `electrics`, first calls the computation function for ionics, before calling on `parab_sovler`, which is the parabolic solver. The parabolic solver uses PETSc to solve the PDE and its specifications are given by the user.

In our case, as mentioned multiple times, it is the conjugate gradient method with a block Jacobi precondtioner.

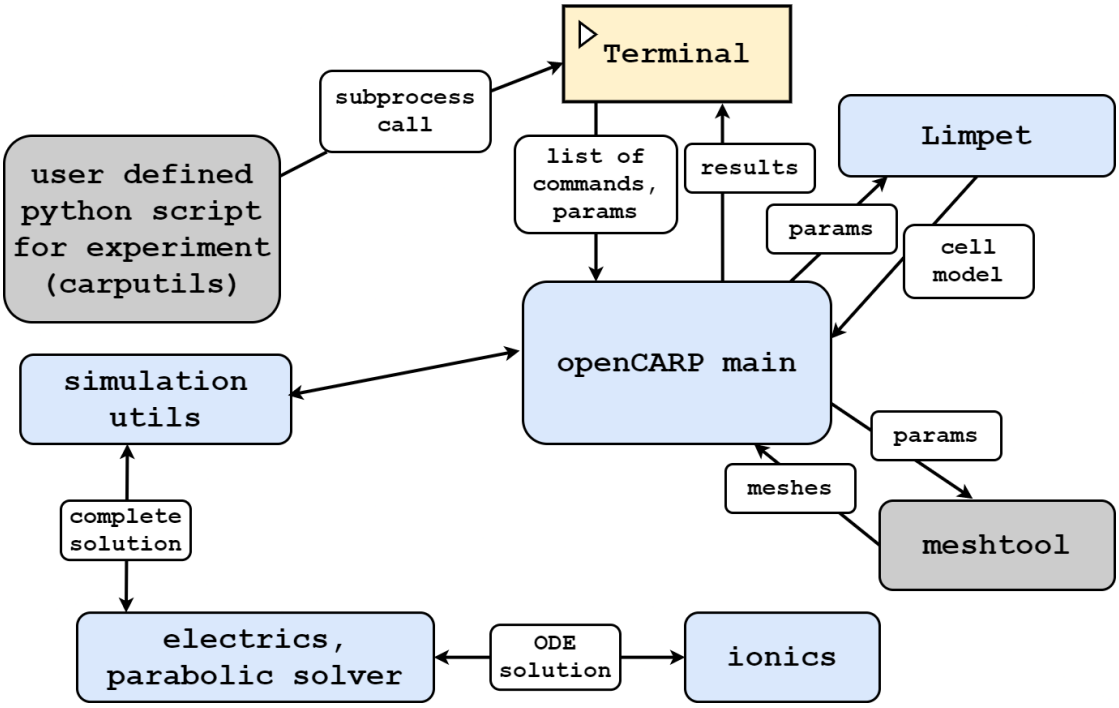


Figure 3.3: Chart showing the architecture of the openCARP solver when used for simulations of the monodomain model. An overview of the main classes, and how they call upon one another. openCARP modules are shown in blue.

The Complete Simulation

The whole simulation is controlled by openCARP's `sim_utils` as well as the `main` script. First the PETSc settings are parsed, and a mesh is either created or loaded. Then, all the physics, electrics, and ionics are registered from the list of commands and initialized. The simulation is run with the `simulate` function, which calls on the `compute_step` function for the `electrics` at each time step. As we have seen, the compute function makes use of the `ionics` and the cell model, as well as the PETSc PDE solver, to complete each step of the solution.

3.4 Main Differences Between the Solvers

In many regards, cbcbeat and openCARP share multiple similarities when it comes to how they do cardiac electrophysiology simulations, which is why they are relevant to compare. There are nonetheless some differences, that potentially can have a large impact on the efficiency, that one should take note of.

For the PDE solver, both softwares uses PETSc to solve each iteration and given the same settings, one would assume that this step should be fairly equal in terms of efficiency. There is however a difference in how the matrices are assembled. cbcbeat uses the DOLFIN function `assemble`, which utilizes the finite element methods build into the core of FEniCS and operates on variables of the UFL type `Form`. openCARP on the other hand uses the core C++ `MatMult` on variables of type `petsc_vector.data`. As the right hand side b is assembled for each time step, this can presumably make a considerable difference.

For the ODE solver the Rush-Larsen scheme is implemented relatively straight forward in both solvers, in C++ with a loop that loops over each node in the mesh and simple integration steps. There is quite a difference though, in the way that all the variables that I_{ion} is dependant on, are calculated. As mentioned, cbcbeat uses UFL expressions, that are evaluated for variables such as V and ion concentrations. The openCARP cell models, have implemented *lookup tables* for all the variables dependant on V and the subspace calcium concentration Ca_{SS}.

Lastly in cbcbeat, though most of the heavy computation is done by the C++ backend, the main time loop itself is done in python, as opposed to openCARP where everything is done in C++. Though, with simulations with computation time of the magnitude one typically encounters in cardiac electrophysiology where there are far more time consuming operations involved for each time iteration, it is unlikely that this should amount to much of a difference.

3.5 Implementation of the Benchmark

For the numerical experiments, each of the solvers were run in a docker [Merkel, 2014]. The cbcbeat[1] docker used for the work is provided by Simula, and the openCARP [2] docker is given by the official webpage.

All code written for this thesis, as well as the raw results, is available in an open repostiroy [3].

To assure that the software is implemented as intended, the case has been coded following examples provided by the developers in both solvers. For cbcbeat, a version of the benchmark was provided as a demo under `cbcbeat/demo/niederer-benchmark`, which has been used as inspiration for our implementation. For CARP, no implementation of the Niederer benchmark was available, but there was however, an example for CARPentry, an older version of CARP[4]. The openCARP case was coded using the CARPentry example as a starting point, and the other openCARP examples and tutorials as inspiration. When one has familiarized oneself with the software and their functionality the implementation is relatively straight forward in both solvers.

Cbcbeat

The core experiment implementation of cbcbeat is `monodomain_benchmark.py` in the *cbcbeat* folder in the repository. First, we define a set of parameters for our benchmark, such as duration, resolution, scheme and solver types, as exemplified in listing 3.1.

[1] Cbcbeat docker: `https://hub.docker.com/r/ceciledc/fenics_cbcbeat`. Pulled with
`docker pull ceciledc/fenics_cbcbeat`

[2] Installation of openCARP: `https://opencarp.org/download/installation`. Docker pulled with
`docker pull docker.opencarp.org/opencarp/opencarp:latest`

(Both dockers were last pulled in January 2021)

[3] Github repository with implemented code and numerical results: `https://github.com/einaj/electrophysiology_benchmark`

[4] CARPentry example of the Niederer benchmark `https://carpentry.medunigraz.at/examples/tutorials/tutorials.02_EP_tissue.11_discretization.run.html#`

```
benchmark_parameters = Parameters("Benchmark")
benchmark_parameters.add("theta", 0.5)  # 2nd order splitting
benchmark_parameters.add("dt", 0.05)    # time resolution
benchmark_parameters.add("dx", 0.5)     # space resolution
benchmark_parameters.add("T", 60.0)     # duration

# conjugate  gradient
benchmark_parameters.add("algorithm", "cg")
```

Listing 3.1: Example of how benchmark parameters can be set to control the experiment

We set up a mesh, define the conductivity tensor and the initial stimuli. The TT2 cell model is imported from the `cellmodel` library. These are again used to initialize the cardiac model. We create a set of parameters for the splitting solver, and add the previously defined benchmark parameters. The splitting solver can now be created, and set up with initial conditions. Finally we can set up the solver generator and solve for each time step. See listing 3.2 for a showcase of how the most important steps are implemented.

```
cell_model = Tentusscher_panfilov_2006_epi_cell()
inits = cell_model.initial_conditions()
...
cardiac_model =  CardiacModel(mesh, time, M, None, cell_model,
   stimulus)
...
ps = SplittingSolver.default_parameters()
    ps["pde_solver"] = "monodomain"
    ps["MonodomainSolver"]["linear_solver_type"] = "iterative"
    ...

solver = SplittingSolver(cardiac_model, ps)
(vs_, vs, vur) = solver.solution_fields()
vs_.assign(inits)
...

solutions = solver.solve((t0, T), dt)
for (i, ((t0, t1), fields)) in enumerate(solutions):
    ...
    ...
```

Listing 3.2: The main steps of the cbcbeat solver implementation. For the full code, see the repository [3].

Any parameters that we will want to change from run to run, such as duration, resolution, whether or not to store the results and name of output directory, have been implemented to be controllable via command line arguments. If `store` is set to *True*, the voltage of every point in the mesh at every time is stored in a .h5 file [The HDF Group, 2021]. For the highest resolutions these can become quite large. If we wish to calculate the activation times, the value of the membrane potential at each point in the mesh is checked, so that if it has surpassed the threshold, the current time is set as that points activation time. `compute_activation_times.py` in the *cbcbeat* folder handles this process.

openCARP

The implementation of the case in openCARP is located in the *carp* folder in the repository, and consists of the initial cell state file `cell_init.sv`, the parameter file `monodomain.par` and the python script `run.py`. Note that openCARP generally operates with μm instead of mm, so some constants might appear to be different from cbcbeat.

The `cell_init.sv` determines the choice of cell model (TT2) as well as its initial values, as given in 3.3. In the .par file, we specify parameters for the solver that we are unlikely to change, such as the conductivities, stimulus, and the choice of model and scheme for the solvers. An example of some command lines we can give in `monodomain.par` is shown in listing 3.3.

```
num_imp_regions = 1
imp_region[0].name = "Ventricle"
imp_region[0].cellSurfVolRatio = 0.14
...
num_gregions = 1
gregion[0].g_il = 0.17
gregion[0].g_it = 0.019
...
bidomain = 0
parab_solve = 1
```

Listing 3.3: Example of command lines found in the parameter file

In the `run.py` we set up the experiment itself. First we define a `JobID`, create the mesh and import the contents of the .par file into the cmd list.

Then we add parameters given in command line such as resolutions, duration, and

name of the output directory. If we aim to measure the activation times as well, the command lines to do so are added to the `cmd` list as well.

The simulation is then run with the `carp(cmd)` function. A rundown of the main steps of the openCARP implementation is displayed in listing 3.4

```
cmd = tools.carp_cmd('monodomain.par')
cmd += ['-meshname', meshname,
        '-simID',    job.ID,
        '-dt',       args.dt,
        ...
        '-imp_region[0].im', 'TT2',
        '-imp_region[0].im_sv_init',"cell_init.sv"]

job.carp(cmd)
```

Listing 3.4: The main steps of the openCARP solver implementation. For the full code, see the repository[3].

If we want to store the state of the membrane at some point t, or calculate the activation times, this is simply done by adding a few more lines to `cmd`. In the openCARP benchmark, the calculation of activation times are handled with if-tests, so that it can be turned on or off with command line arguments without the need for a second script.

3.6 Verification

To verify that our implementation of the case is accurate, we reproduce the simulation protocol from the Niederer benchmark for every combination of the resolutions $\Delta x = [0.5, 0.2, 0.1]$ and $\Delta t = [0.05, 0.01, 0.005]$ ms, and compute the activation times.

3.7 Timing

When benchmarking the efficiency of the two solvers, we want to measure the wall time as they execute the simulation. In order to properly measure the computation time of the core simulation, we choose not to store the solution at any point, or calculate the activation times.

As will be shown in chapter 4, the activation time at P8 is a lot earlier for the higher resolutions. The zero flux boundary conditions can make the solution unstable if the simulation runs for a long time after the P8 activation. Therefore, we want to choose an end time T at some point before that happens. Furthermore, wall time can become very long for higher resolutions if T is too large. Choosing $T = 30$ ms is fitting, as it is not too long, but long enough so that the quickest simulations still have a consistent wall time.

The script `time_benchmark.py` is made as a method of making the timing process easy and efficient.

Each of the solvers are timed multiple times for all resolution configurations, and some resolutions are also run in parallel for 2, 4 and 6 threads using MPI. Then, the average wall time is calculated. When finding the average wall time for a set of runs, the measured time of the first run is excluded, as it has a tendency to be slower due to just-in-time compilations for cbcbeat, or mesh generation in the case of openCARP.

Lastly, as a method of dissecting the solvers and identify the different steps' contribution to the total wall time, we use the solvers' built-in timers to give us a detailed overview for some selected runs. cbcbeat provides a detailed table of the different operations, showing how many times they are repeated, the average time per execution, as well as their contribution to the total wall time. Setting the output level of openCARP to 5, we can get quite a lot of details, though, not as extensive as for cbcebat.

Chapter 4

Results

In this chapter, we present the results of the simulations produced by our implementation and timing of the Niederer benchmark, as described in chapter 3. All results are created using an Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz 6 core processor, and 32 GB of 3000Mhz RAM.

4.1 Verification of accuracy

The activation times produced by our implementation of the Niederer benchmark in cbcbeat and openCARP are presented in table 4.1 and 4.2. For a visual presentation of the propagation along in the tissue, simulated in cbcbeat for $\Delta x = 0.1$mm and $\Delta t = 0.05$ms, see figure 4.6.

For both solvers, it is clear that the real jump towards the converged 'true' solution happens whenever the spatial resolution is increased. The difference between the activation times for a point calculated at the same spatial resolution only varies about 0.1% - 4.2% for the different temporal resolutions. Increasing the spatial resolution from 0.5 to 0.2 mm, on the other hand, can almost halve the activation time at certain points. It also appears that $\Delta x = 0.2$mm comes quite close to the converged solution, so increasing further to 0.1 mm does not grant nearly as much improvement as the step form 0.5 to 0.2 mm.

cbcbeat activation times (ms)									
Δx, Δt	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_C
0.1,0.005	1.225	31.085	7.965	31.550	25.400	37.620	25.995	37.735	17.750
0.1,0.01	1.230	31.170	7.980	31.640	25.460	37.720	26.04	37.840	17.800
0.1,0.05	1.300	31.850	8.150	32.300	25.850	38.500	26.450	38.600	18.150
0.2,0.005	1.225	30.915	9.285	30.950	28.990	38.355	29.280	38.110	17.725
0.2,0.01	1.230	30.990	9.300	31.020	29.000	38.410	29.300	38.170	17.760
0.2,0.05	1.250	31.550	9.350	31.600	29.150	38.900	29.400	38.650	18.025
0.5,0.005	1.215	33.775	13.850	32.890	50.520	57.885	49.285	55.665	25.555
0.5,0.01	1.220	33.810	13.920	32.920	50.680	57.910	49.290	55.710	25.570
0.5,0.05	1.250	34.050	14.000	33.150	50.800	58.150	49.250	55.950	25.700

Table 4.1: Activation times for cbcbeat, at the nine P points as defined in figure 3.1b). The solution converges towards a P8 activation of about 37.5 - 38 ms.

openCARP activation times (ms)									
Δx, Δt	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_C
0.1,0.005	1.252	31.268	7.781	32.257	25.258	39.423	25.907	40.074	18.706
0.1,0.01	1.264	31.427	7.816	32.422	25.352	39.619	26.006	40.273	18.804
0.1,0.05	1.365	32.627	8.087	33.657	26.058	41.085	26.751	41.750	19.544
0.2,0.005	1.252	30.845	8.723	31.858	28.336	40.165	28.855	40.461	19.187
0.2,0.01	1.264	30.981	8.741	31.998	28.363	40.302	28.882	40.601	19.263
0.2,0.05	1.365	31.987	8.881	33.032	28.561	41.352	29.082	41.668	19.863
0.5,0.005	1.251	32.895	11.631	33.815	47.755	58.250	48.049	58.337	24.943
0.5,0.01	1.262	32.949	11.644	33.870	47.762	58.361	48.057	58.447	25.003
0.5,0.05	1.363	33.331	11.730	34.257	47.708	59.143	48.009	59.223	25.450

Table 4.2: Activation times for openCARP, at the nine P points as defined in figure 3.1b). The solution converges towards a P8 activation of about 40.0 ms.

4.2 Execution time

4.2.1 Serial Execution: Total Wall time

The average execution time of the two solvers given certain resolutions are presented in table 4.3

Average Wall Time			
Δx, Δt	cbcbeat (seconds)	openCARP (seconds)	cbcbeat/openCARP
0.1,0.005	14208.70	1634.03	8.70
0.1,0.01	7035.56	794.06	8.86
0.1,0.05	1454.42	186.18	7.81
0.2,0.005	1825.43	209.25	8.72
0.2,0.01	895.15	107.05	8.36
0.2,0.05	177.80	23.58	7.45
0.5,0.005	132.11	13.60	9.71
0.5,0.01	66.34	7.41	8.96
0.5,0.05	14.58	1.91	7.63

Table 4.3: Average wall time for the benchmark simulation for the two solvers as well as the relative difference between them, for all combinations of the resolutions. All simulations are made with $T = 30$ s.

On average, openCARP computes the simulations of the same experiment around 8-9 times as fast as cbcbeat, which is a significant difference in performance. The biggest difference between them occurring for the highest temporal resolution, indicating that cbcbeat performs worse as the temporal resolution is increased. This phenomenon is easier to see when the average times for a given spatial resolution are plotted as functions of the temporal resolution, as seen in figure 4.1. For the increase in spatial resolution, however, there is no obvious difference in performance between the two solvers, which is evident in figure 4.2

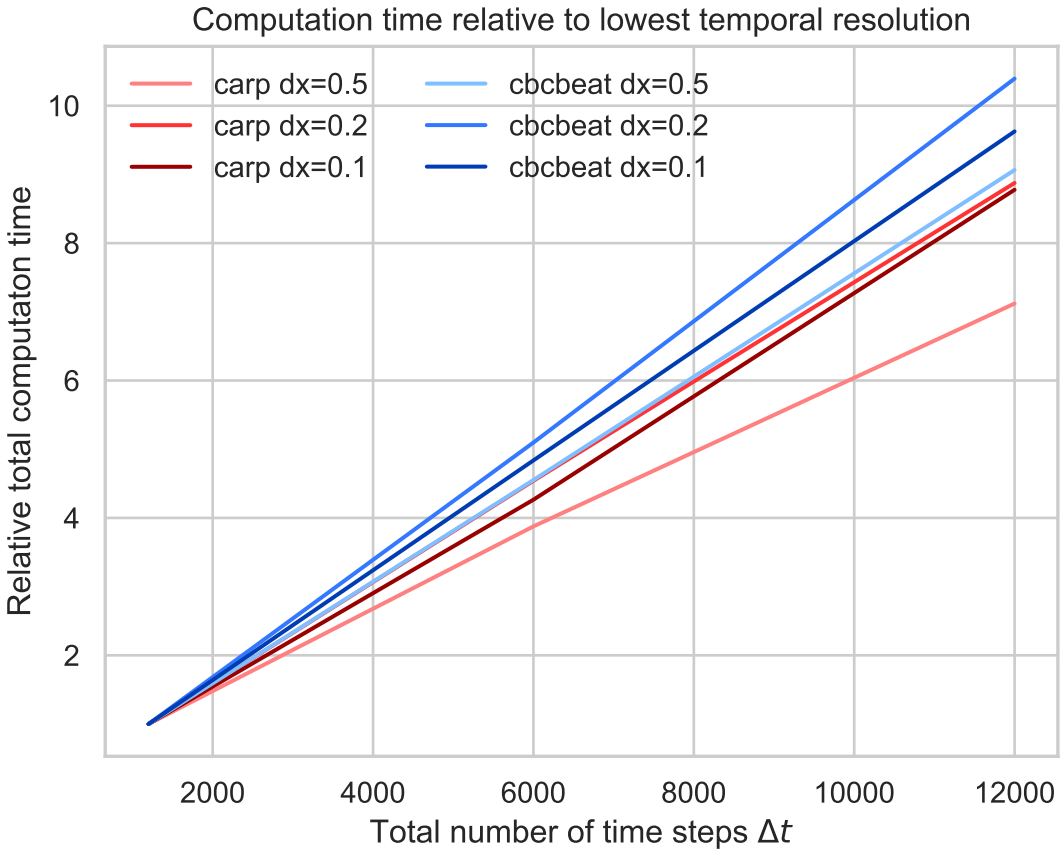


Figure 4.1: The average execution wall time of cbcbeat and openCARP for the three spatial resolutions as a functions of the temporal resolution. The plots show the relative computation time, that is, the wall time for a given Δt divided by the wall time for the lowest temporal resolution $\Delta t = 0.05$ ms. Both solvers' wall time increase linearly with the amount of time steps, but cbcbeat has steeper lines, indicating that carp handles the increase in temporal resolution better.

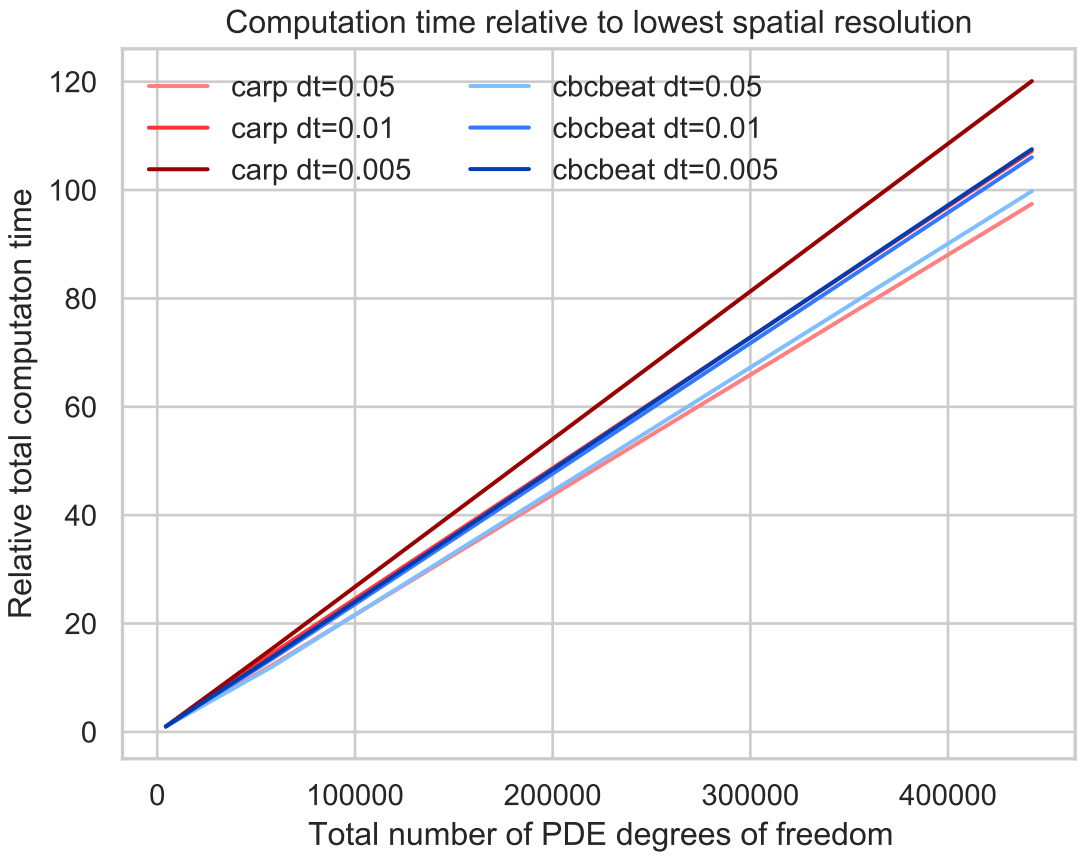


Figure 4.2: The average execution wall time of cbcbeat and openCARP for the three temporal resolutions as a functions of the spatial resolution resolution by the number of degrees of freedom for the PDE. The plots show the relative computation time, that is, the wall time for a given spatial resolution divided by the wall time for the lowest spatial resolution $\Delta x = 0.5$, PDE-dofs $= 4305$. Both solvers' wall time increase linearly with the number of degrees of freedom, and although there is some difference between the different runs, there is no clear distinction between the cbcbeat and openCARP.

4.2.2 Serial Execution: Detailed Insights

cbcbeat

For $\Delta x = 0.2$ mm, $\Delta t = 0.05$ ms and $T = 30$ s, an example run of cbcbeat used 179.8 seconds. The main parts' contribution to the total wall time are presented in table 4.4. For even more details for this run, as well as detailed timings for a parallel run, see appendix A.1.

Detailed Timings of cbcbeat			
Operation	repetitions	avg. wall time [s]	total wall time [s]
ODE step	1200	0.10137	121.64
PointIntegralSolver::step	1200	0.10012	120.14
PointIntegralSolver::apply	1200	3.9632e-05	0.047558
PDE Step	600	0.092932	55.759
Assemble rhs	600	0.084973	50.984
PETSc Krylov solver	600	0.0077343	4.6406
SplittingSolver: setup	1	2.1359	2.1359
Init dofmap	3	0.2462	0.73859
Merge step	600	0.0013864	0.83186

Table 4.4: Detailed insights in the wall time used by the different steps in the cbcbeat solver, showing the number of times a step is applied, the average wall time, and the total wall time contribution from each step. All times given in seconds.

With 600 time steps, the ODE part of the equation is solved 1200 times in which it contributes with 122 seconds, or about 67 % to the total computation time. The point integral solver is the only non-negligible contributor to the ODE solution step time. The PDE, which is solved 600 times, contributes with about 56 seconds (30 %), where 51 seconds is the rhs matrix assemble process and 4.6 seconds are used by the PETSc Krylov solver. The remaining few % of the total wall time are distributed over smaller processes such as splitting solver setup and the merge step. A sunburst chart showing the how the total computation time is distributed over the steps are shown in figure 4.3

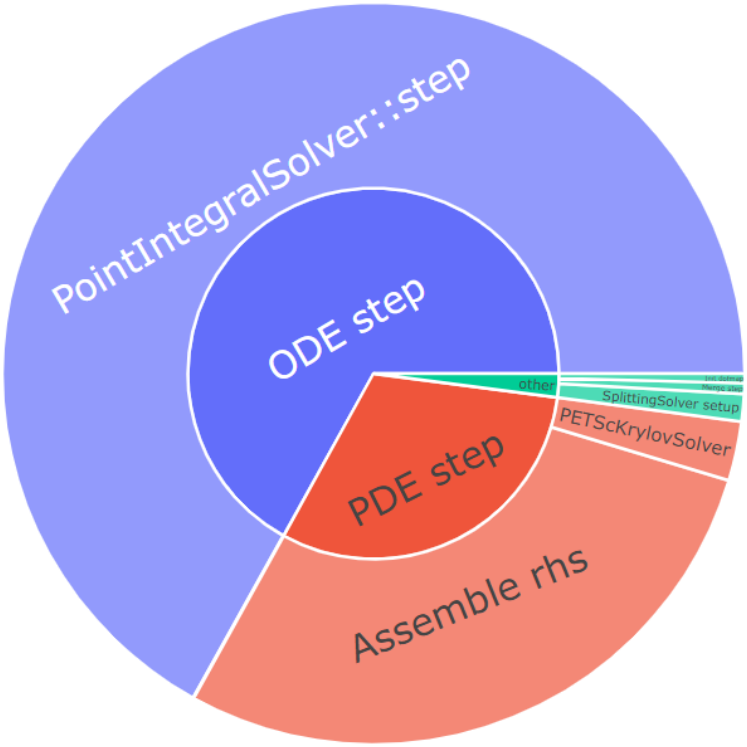


Figure 4.3: A sunburst chart showing the contributes to computation time in cbcbeat, with the ODE step, PDE step, and other contributions are divided into sub-operations.

openCARP

The information provided about the detailed timing from openCARP takes a somewhat different form than that from cbcbeat. Using the same parameters as for cbcbeat, $\Delta x = 0.2$ mm, $\Delta t = 0.05$ ms and $T = 30$ s and serial execution, the total wall time for openCARP for an example run is 23.81 seconds. The time spent by the main operations are shown in table 4.5. For even more details for this run, as well as the timings for a parallel run of 4 MPI threads, see appendix A.2.

As mentioned in section 3.3, *electrics* and *ionics* in openCARP refer to the PDE step and the ODE step respectivly. As such, the PDE computation is resposnible for the most substantial chunk of the computation time, using roughly 54% of the total wall time. The ODE spends about half of that time. The sunburst chart in

Detailed Timings of openCARP	
Operation	time [s]
Setup meshes	3.93
Initiate Electrics	0.44
Initiate Ionics	0.08
Compute Electrics	12.85
Compute Ionics	6.23

Table 4.5: Detailed insights in the wall time used by the different steps and components of openCARP. All times given in seconds

figure 4.4 gives an overview of the how the total computation time is partitioned between the different operations. When comparing cbcbeat and openCARP, one should probably focus on the numbers presented in table 4.4 and 4.5. Comparing figure 4.3 and 4.4 might give a false impression of very slow initialization in openCARP, compared to cbcbeat, but this is mainly due to openCARP spending considerable less time on the solution steps, making them more equal in size. It is also important to note that though openCARP spends a lot of time that cbcbeat does not on setting up meshes, this cannot be directly compared as it is handled differently in the two solvers. openCARP uses *meshtool* to create or load previously created experiment specific meshes, while cbcbeat imports build-in DOLFIN meshes.

4.2.3 MPI parallelization

Using MPI, we run both cbcbeat and openCARP using 2, 4 and 6 parallel threads for a selection of the resolutions. The results are presented in figure 4.5.

As the number of threads increase, cbcbeat generally appears to achieve close to maximum speedup, with the relative computation time curve flattening out around 4-6 threads. The higher spatial resolution, the more cbcbeat benefits from parallelization, reaching a total wall time of less than 25 % of the serial execution for $\Delta x = 0.1$mm, $\Delta t = 0.05$ ms for 6 parallel MPI threads. openCARP initially achieves a speed boost, reaching about 65% of its serial computation time at 2 and 4 parallel MPI threads, and down to 50% for selected parameters. For 6 threads however, the speedup is reduced, becoming as slow as serial execution for certain resolutions.

In appendix A.1 and A.2, the listings A.2 and A.4 show the detailed operation

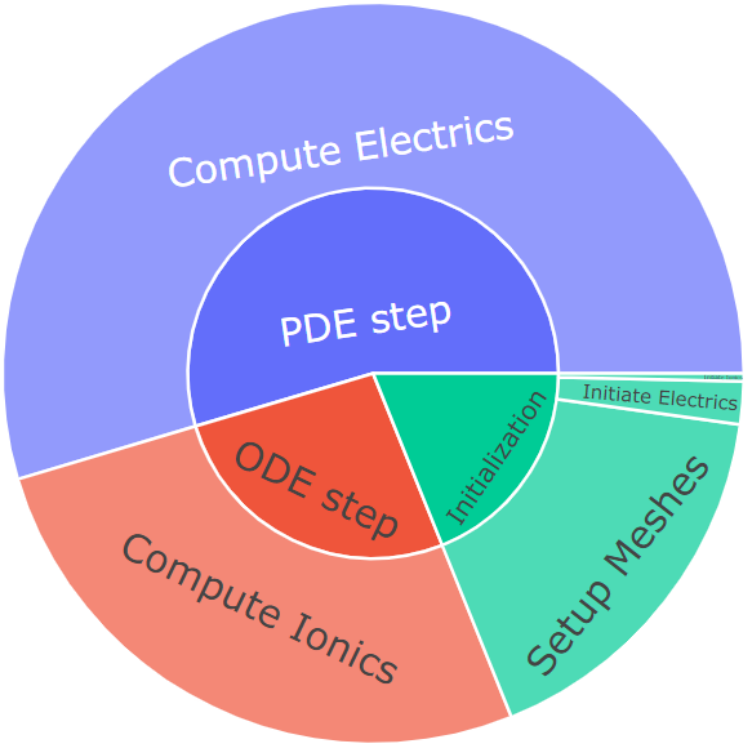


Figure 4.4: A sunburst chart showing the contributes to computation time in openCARP, with the ODE step, PDE step, and the initialization divided into sub-operations.

specific timings for parallel runs of the solvers, given the same parameters as discussed above. They are timed for their optimal number of threads, which is 6 for cbcbeat and 4 for openCARP. Many of the operations related to initialization and merging are relatively unchanged, but others change quite drastically. The most interesting timings are showed in table 4.6. The detailed insights are shown for the same resolutions as the serial execution times above, that is $\Delta x = 0.2$ mm, $\Delta t = 0.05$ ms, $T = 30$ s.

For cbcbeat, the ODE solver computation time is drastically sped up, as the step function is reduced to only a fifth of its original wall time. For the PDE, the solver itself is more or less unchanged, but the assemble process appears to benefit greatly from parallelization, spending only 22 % of the time it did for serial execution. openCARP provides slightly less details, but much like for cbcbeat, the ODE step (ionics) is reduced to less than 30 % percent compared to the serial execution wall time. The PDE also benefits somewhat of the MPI parallelization,

but not nearly as much.

Parallel vs Serial Individual Operation Timing			
Operation	serial time [s]	MPI time [s]	MPI/serial
cbcbeat			
PETSc Krylov solver	4.6406	4.7126	1.02
Assemble rhs	50.984	11.151	0.22
PointIntegralSolver::step	120.14	22.691	0.19
openCARP			
Compute Electrics	12.85	11.31	0.88
Compute Ionics	6.23	1.74	0.28

Table 4.6: The timing of a selection of solver operations for serial execution vs optimal MPI parallelization.

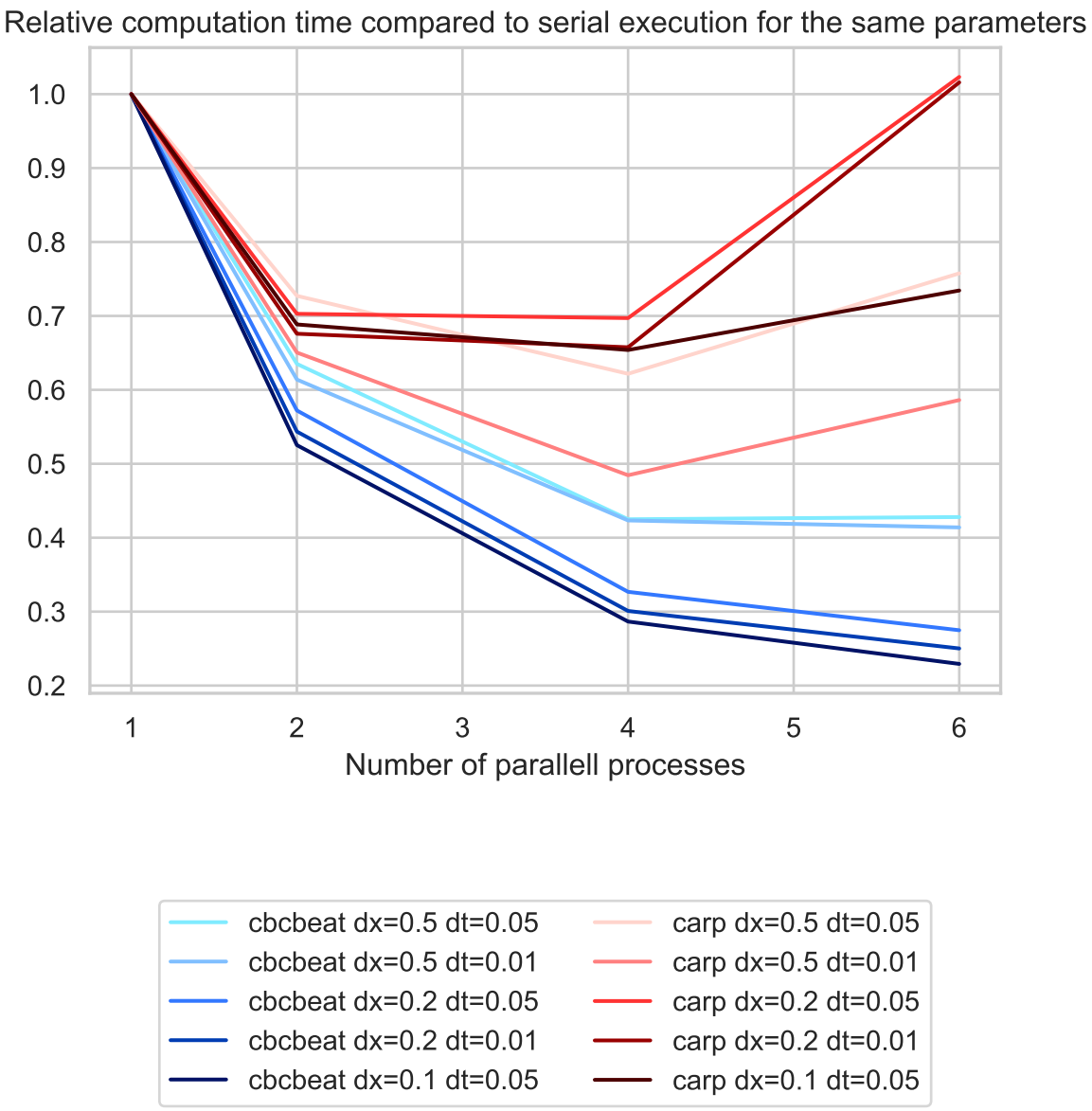
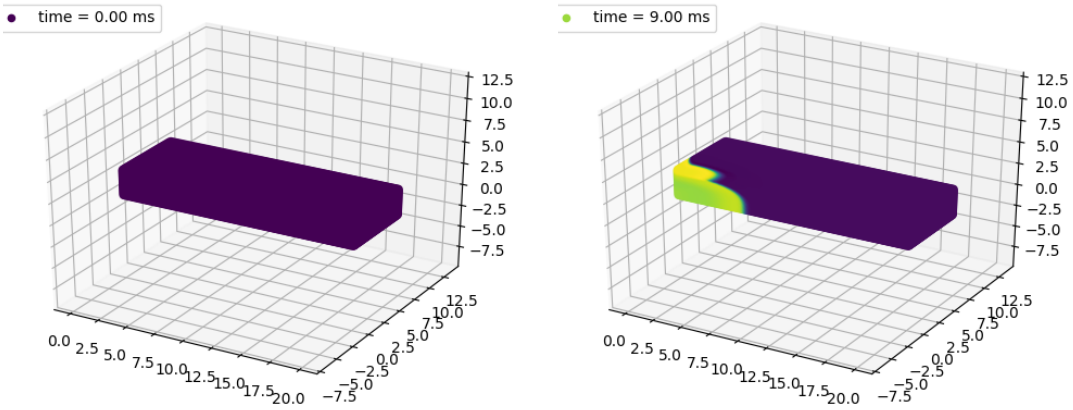
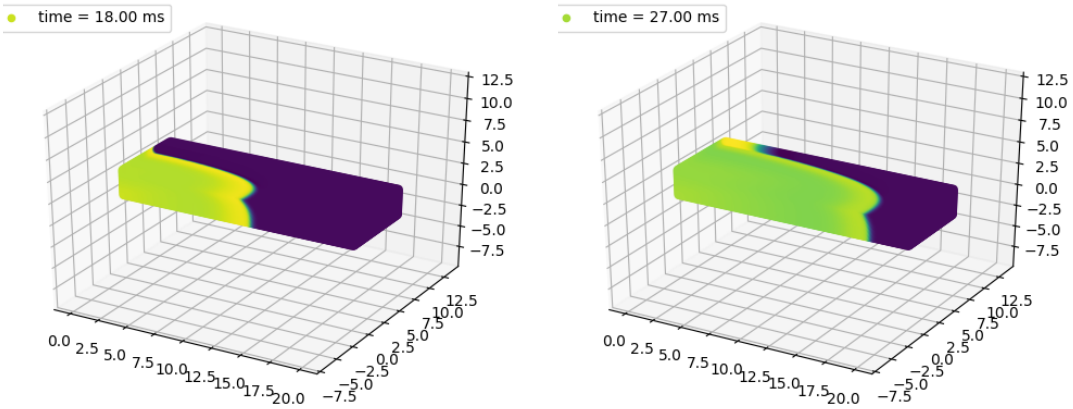


Figure 4.5: Plot showing the relative execution time for different configurations of the two solvers, with number of parallel threads along the x-axis. The wall time for n threads is divided by the serial execution wall time for the same configuration to display the speedup achieved by parallelization.



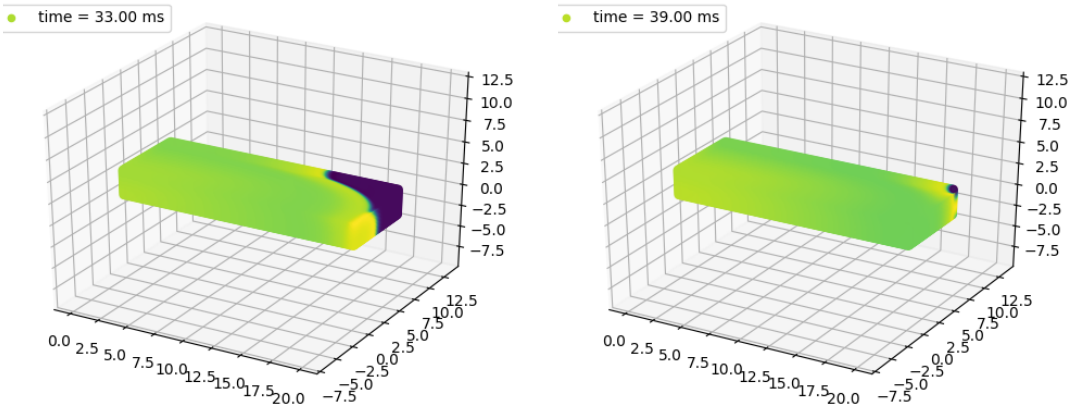
(a)

(b)



(c)

(d)



(e)

(f)

Figure 4.6: Simulation of the potential spreading through the tissue slice, made with cbcbeat using $\Delta x = 0.1$mm and $\Delta t = 0.05$ms. Plots at times (a) 0 ms, (b) 9 ms, (c) 18 ms, (d) 27 ms, (e) 33ms and (f) 39 ms.

Chapter 5

Discussion

5.1 Verification and Accuracy

The activation times of the two solvers for the P1-P8 and center points presented in table 4.1 and 4.2 correspond well with the results that are presented in the supplemental material of the Niederer benchmark, [Niederer et al., 2011], and we can consider the implementation of the the benchmark problem in both solvers verified. The results of cbcbeat in particular are very similar to what FEniCS originally produced in the Niederer article, and converges towards a P8 activation time of 38 ms, which is slightly below the 'true' solution, at least as Niederer et. al. defined it ($\sim$43 ms). openCARP also converges to activation times within the spectrum of activation times from the original benchmark, however, there are some noticeable deviation from the CARP results from 2011. Firstly the P8 activation time for the highest resolution is reduced by about $\sim$13 % (46 ms $\rightarrow$ 40 ms). Secondly the convergence has increased quite drastically, as the Niederer results for CARP measured a P8 activation time at as much as 126 ms for the lowest spatial resolution, while our implementation results in an activation time of about 59 ms for the same resolution.

The difference in the convergence rate of some of the solvers are discussed in the Niederer article, where it is suggested that it is due to some of the solvers *lumping the mass matrix* while others do not. To verify this, our openCARP implementation can be run using mass lumping. In openCARP this is done by simply changing an input parameter, and the resulting activation times are shown in table 5.1.

cbcbeat activation times (ms)									
Δx, Δt	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_C
0.2,0.05	1.36	34.09	9.50	35.68	31.40	47.00	32.29	47.66	21.91
0.2,0.05	1.36	36.36	12.29	38.98	40.87	56.30	41.92	56.79	26.46
0.5,0.05	1.37	49.69	24.87	55.42	104.68	118.60	105.78	118.10	51.58

Table 5.1: Activation times for openCARP for a selection of resolutions, using a lumped mass matrix.

It is clear that not using a lumped mass matrix can account for the increased converge rate in our implementation compared to the Niederer benchmark results for CARP. When it comes to the reduced activation time for the converged P8 solution, the use of a lumped mass matrix in our simulations, bring the $\Delta x = 0.1$ activation time of P8 for openCARP up to its original 2011 value. The discrepancy between our implementation of openCARP and the CARP activation times presented in the Niederer benchmark can therefore be explained almost fully by the use/non-use of a lumped mass matrix. Additionally, CARP is a software that has been in continuous development over the last 10 year, openCARP being only the latest instalment, so there is a possibility that some changes to the implementation of the solver backend during development has happened as well.

5.2 Efficiency of the Solvers

5.2.1 Total Wall Time

As the results in table 4.3 show, openCARP outperforms cbcbeat with about 8-9 times in terms of efficiency when simulations are computed with serial execution, which is quite a staggering difference. From table 4.3, as well as figure 4.1 and 4.2, we can get an idea of how the solvers overall performance scales in terms of temporal and spatial resolution. Evidently, the difference in efficiency increases for higher time resolutions, and as seen by the steepness of the plots in figure 4.1, cbcbeat's relative computation time increases more for an increased number of time steps than wat openCARP does. This indicates that the increase in temporal solution is handled better by openCARP than cbcbeat.

5.2.2 Individual Operation Times

The detailed insights provided by the built-in timer functions in the two solvers provide valuable information about the various operations needed to complete the simulation, and how they compare in terms of their contribution to the total wall time. The results presented in table 4.4 and 4.5 reveal some very interesting properties and differences between the two solvers.

For the PDE step, the two main operations is the solution step, using the conjugate gradient method, and the assembly of the right hand side vector. As both solvers use PETSc for the solve step, we should expect them to be approximately equal in this regard. cbcbeat spends ∼56 seconds on the whole step, but out of this, only 4.6 seconds is the PETSc solver. Thus, the PDE step in cbcbeat is fully dominated by the assembly process, which contributes with 91% of the total PDE computation time. For openCARP the timings are slightly less detailed, but we can make the assumption that the solve part of the PDE is roughly equal to that of cbcbeat, which is likely as they both use PETSc with the same settings. As the total electrics/PDE step in openCARP spends ∼13 seconds, it leaves about 8 seconds for right hand side vector assembly and/or other operations. In other words, it becomes clear that assembling the right hand side of the PDE (2.54) is a lot more optimized in openCARP than it is in cbcbeat.

In cbcbeat, the total ODE step is responsible for the largest amount of computation time by far, as seen in figure 4.3, where the point integral again is the sole non-negligible component. This stands in stark contrast to openCARP, where as figure 4.4 displays, the ODE step accounts for a more modest ∼25 % of the total wall time. The 120 seconds spent by the solver in cbcbeat are mammoth compared to the meagre 6.23 seconds that it takes openCARP to compute the same operation.

As mentioned in chapter 3, this is one of the areas in which the implementation of the two solvers differ quite substantially. Although they both use the Rush-Larsen scheme, the approach used to calculate the I_{ion} current is very different. cbcbeat uses a set of UFL expression to describe all the 18 state variables that I_{ion} is dependant on, as well as the sub-variables that they again depend on. These expressions must at some point at each step be evaluated for variables such as V and calcium concentration. For the same calculation openCARP uses ready-made lookup tables that are created when a cell model is constructed. The calculation of the ion current must be done at each time step, so the speed up gained from efficiently calculating all the variables can potentially be massive. This is especially true when more intricate cell models are used, as they include a large set of variables with potentially complex and computationally expensive

function calls.

As more detailed timings from inside the ODE solver is not available for either of the solvers, it is somewhat difficult to determine exactly how much of the difference in efficiency is accounted for by the use of lookup tables versus expression evaluations. An approach to gain some more insight into this, is to run both solvers for a much simpler cell model, with far fewer and less complex expressions. In such an experiment, the simplicity of the model means that the computation of I_{ion} becomes a much less significant part of the total solve step, and the difference in wall time will depend more on the efficiency of the other parts of the solve step, such as looping over the nodes and performing the integration. An example of such a cell model is the Aliev-Panfilov [Aliev and Panfilov, 1996], which only has one state variable in addition to V, and that we introduced in section 2.2.2. With the way we have implemented the two solvers, changing the cell model is done by a single line of code, and allows us to run the exact same experiment, just with a different cell model. The timing of the central solve steps when using Aliev-Panfilov instead of ten Tusscher & Panfilov is shown in table 5.2.

Complex vs Simple Cell Model Timing		
Operation	TT2 time [s]	AP time [s]
cbcbeat		
PETSc Krylov solver	4.640	4.286
Assemble rhs	50.984	37.122
PointIntegralSolver::step	120.14	18.597
Total wall time	176.6	62.22
openCARP		
Compute Electrics	12.85	11.49
Compute Ionics	6.23	0.12
Total wall time	23.81	15.6

Table 5.2: The timing of a selection of solver operations for the TT2 model vs a simple cell model Aliev-Panfilov (AP).

For openCARP, the PDE/electrics step computation time is slightly reduced for the simple cell model, however the change is relatively small. For the ionics/ODE step on the other hand the computation time is reduced to practically nothing, about 2% of the TT2 computation time. The total wall time is reduced to ∼65 %.

For cbcbeat, the PDE solver time is more or less the same, but the assemble process time is reduced by a noticeable amount. The biggest difference is definitely in the

ODE step, where the solver's computation time is reduced to just 15 % of the time it spent when using the TT2 model. The ODE time is reduced to such a degree, that for the simple AP cell model, the PDE step is suddenly the main contributor to the total wall time, which in turn is reduced to ∼35 % of the execution time of the more complex TT2 model. For the total execution time, the change to a less complex model benefits cbcbeat a lot more than it benefits openCARP. As everything else is the same, the large drop in the ODE computation time can be attributed to the calculation of I_{ion}.

Even though the simple AP model using only a single state variable should reduce the benefits from the use of lookup tables in openCARP, there still remains a huge difference in the ODE computation time. This points to there being additional factors that make the openCARP ODE sovler so much faster. Possible reasons could be the implementation of the integration step, or the choice of variable type, which in cbcbeat often is defined by FEniCS types and UFL, while openCARP uses more traditional C++ vectors, and often PETSc types.

5.2.3 Parallelization

Figure 4.5 shows how the two solvers relative computation time change for 1,2,4, and 6 parallel MPI threads for a selection of resolutions.

In general, it is clear that cbceat has more to gain from parallelization than what openCARP has. For cbcbeat, it appears that the higher the spatial resolution, the more it benefits, with the $\Delta x = 0.1$ mm, $\Delta t = 0.05$ ms, run having it's computation time reduced to less than 25% of the serial execution time for 6 threads. openCARP also gains a significant speed boost from parallelization, though less than cbcbeat. The openCARP run with the lowest total wall time reaching ∼50%, and the run with the highest minimum only reaching 70 % of serial time. For 6 cores, all the openCARP runs starts gaining wall time again, probably due to the overhead of parallelization taking over. For the simulations done on the processor used in this thesis work, it seems that 4 parallel threads are optimal for openCARP, while the full utilization of all 6 threads is optimal for cbcbeat. Though there might be properties of the cbcbeat implementation that makes it more susceptible to parallelization, we must also take into consideration that the openCARP runs are a lot faster in general, and will by that nature have less to gain from parallelization. This could potentially be explored by repeating the parallelization for an up-scaled, larger problem on which openCARP would spend more time.

Table 4.6 shows more detailed insight into the computation time for serial and par-

allel execution for the optimal number of threads for each solver. For cbcbeat, the PETSc solver is unchanged, while both the PDE right-hand-side vector assembly and the ODE integration step is reduced to roughly 20 % of their serial execution time. Something similar is observable also for openCARP, where the ODE/ionics step is reduced to 28 % of the serial computation time. As we do not have the detailed enough information to know what has contributed to the 12 % reduction in wall time, we cannot pinpoint exactly where the time is saved. However, if we assume that the openCARP PETSc solver will behave similarly to that of cbcbeat and that the solve time will not change notably, then we can attribute the speed boost achieved by the electrics/PDE step to matrix assembly or other operations within the *compute electrics* step.

5.3 Limitations of the Work

5.3.1 Execution in Dockers and Limited Timing Insights

The developers behind both cbcbeat and openCARP encourage users to run the software from docker containers to provide consistent environments and avoid issues that can occur when installing from source. Dockers are an elegant and easy to use option, however it has lead to a few limitations regarding the thesis work.

Firstly, the simulations have had to be executed in two different dockers that will have slightly different software installed that could potentially affect the results. Although we assume this to have marginal impact, if any at all, it should be taken note of.

Secondly, running the software from dockers, makes the task of changing the source code somewhat more difficult. cbcbeat generally provides a satisfying level of detailed timing of the solver components, but for openCARP, slightly more timing details would have been favorable. The time results produced in this thesis work are all a result of using the highest possible output level for openCARP out of the box. If more timings were to be done, it would have to be changed in the source code. To achieve this, we would have to build a new, alternative docker for openCARP which was not done due to time constraints. As such, the information regarding the time used by the different components of the computation steps of openCARP was not as detailed as cbcbeat. A consequence is that some of the assumed behaviour of openCARP is based on assumptions of the PETSc PDE solvers behaving similarly, even though we have no direct proof of this.

5.3.2 Dolfin-adjoint Incompatibility

cbcbeat was orginally written to be compatible with *dolfin-adjoint/libadjoint* [Farrell et al., 2013], a software for automated derivation and computation of adjoint and tangent linear solutions. However, libadjoint latest version is compatible with dolfin 2017.2.0, as the new version *dolfin-adjoint/pyadjoint* [Mitusch et al., 2019] has taken its place. The current cbcbeat docker is running dolfin 2019.2.0.dev0, but is coded to be dependant on libadjoint and not pyadjoint. As such cbcbeat and libadjoint are dependant on different versions of dolfin that cannot be run at the same time. The solution to the incompatibility issue was to turn off the adjoint features in cbcbeat when implementing our experiments. It is possible that the efficiency of cbcbeat might have looked somewhat different, had we been able to utilize the dolfin-adjoint features.

5.4 Further Work

5.4.1 Further Benchmarking

It would be useful to extract even more information about the detailed timings, particularity of openCARP. One can get valuable insight into the theoretical efficiency of the solvers based on the implementation as seen in the open source code, but the actual qualitative differences can only be found through measurement.

Furthermore, we have in this thesis performed an efficiency benchmark of only two of the cardiac electrophysiology softwares that were included in the 2011 Niederer accuracy benchmark. As computation time can be a substantial bottleneck for simulation research, any such software that is actively used in research could benefit from a similar benchmark. A thorough comparison of the efficiency of different solvers, i.e. the ones included in the Niederer benchmark, would be very useful, as it could provide ideas for software improvement across multiple implementations. Additionally, the information about the efficiency of various solvers could prove valuable to someone looking to perform cardiac electrophysiology simulations by choosing from already existing software.

5.4.2 Improvement of Cbcbeat

The obvious extension of the work done in this thesis, is to target the components of cbcbeat that proved to be less-than-optimal, in order to improve the efficiency, and by that the usability of cbcbeat.

Based on our simulations, the most relevant parts to target would probably be:

- The ODE solver. The ODE step in cbcbeat is the largest contributor to the total wall in, as opposed to openCARP, where it takes the backseat compared to the PDE. As we have discussed, this step consist of two main components, and both could potentially be more optimized.

  Firstly, the calculation of I_{ion} and the state variables. The large speed boost cbcbeat experienced when tested for a simpler model indicates that a lot of computation time is spent on calculation the variables of the more complex models. In this case, one might take inspiration form openCARP and consider using lookup tables for calculating many of the variables to avoid expensive function calls or expression evaluations.

  Simultaneously, openCARP outperforms cbcbeat by a staggering amount also for a simple cell model, being ~ 150 times faster for Aliev-Panfilov (see table 5.2). This points to there being some significant overhead in cbcbeat, that cannot be explained by the handling of the ion current expression alone. The DOLFIN functions utilized by cbcbeat such as PointIntegralSolver, are not as much used and tested, and therefore not optimized at the same level as openCARP. Therefore, without giving any specific recommendations, a general optimization of the DOLFIN functions used by cbcbeat, have the potential to increase the efficiency of cbcbeat substantially.

- The assembly of the right hand side vector of the linear PDE. The solve step for the PDE, the PETScKrylovSolver appears to be very well optimized, and calculates the solution with an efficiency similar to openCARP. Where they differ drastically for the PDE is that the assembly process in cbcbeat takes more time than the entire computation step in openCARP, regardless of parallelization. It is therefore clear that also the assembly in openCARP is a lot more optimized than in cbcbeat, and that this is an area of potential improvement. With the current implementation, cbcbeat uses the assemble function from DOLFIN, and a possible approach would be to either improve upon that, or build a cbcbeat-specific assemble function, made to fit the problems for which cbcbeat is used.

Chapter 6

Conclusion

The primary goal of this thesis was to thoroughly compare the two cardiac electrophysiology solvers cbcbeat and openCARP, particularly in terms of efficiency.

This was done by reproducing the experiment outlined in the Niederer accuracy benchmark from 2011, and performing detailed timings of the different steps that make up the solution process in the two solvers. Our implementation of the experiment was verified by comparing the activation times in the corners of the simulation domain to the results for the solvers in the Niederer article. Our cbcbeat results were practically identical to the original results. openCARP deviated a little bit from the CARP results presented originally, however, testing our simulation for a lumped mass matrix proved that the difference could be explained fully by the choice to use/not use mass matrix lumping.

The results from our efficiency benchmark showed that openCARP outperforms cbcbeat by 8-9 times when run in serial execution. Our results indicate that cbcbeat appears to benefit more from parallelization than openCARP, at least for a problem of the size tested in this thesis, with openCARP being about 2.5-3 times faster when run in parallel.

The detailed timing of the various solution steps showed that for serial execution, the ODE solver of cbcbeat is responsible for $\sim$67% of the total wall time. This stands in contrast to openCARP, where the ODE solution time takes the backseat and contributes with just 25 % of the total computation time. It is almost 20 times as fast as cbcbeat when timed for serial execution. Further testing confirmed that cbcbeat's total computation time is dominated by a sub-optimal ODE solver that spends a lot of time computing the ion current term of the monodomain equation,

as well as being subject to overhead when calculating the integration steps.

The PDE solver in both cbcbeat and openCARP utilized PETSc [Abhyankar et al., 2018] well, and is probably as good as optimized. When assembling the right hand side vector of the linear PDE, however, openCARP only spends $\sim$15 % of the time cbcbeat does on the same operation, making it evident that the assembly functionality in cbcbeat also leaves room for improvement.

Having studied the architecture of the two softwares, as well as produced qualitative measurements of their computation time for a given problem, we are able to point to some specific components in cbcbeat and openCARP in which their implementation differ in ways that visibly affect their efficiency. As computation time keeps being a bottleneck for simulation based research, it is in our interest to be able to optimize software as much as possible. By using the insight gained in this thesis we can point out some potential areas in cbcbeat, where one could consider making changes to improve its efficiency, and thereby its usability.

Appendix A

Full Details of Computation Time of the Individual Solver Components

A.1 Detailed Timings of cbcbeat

[MPI_AVG] Summary of timings	reps	wall avg	wall tot
Apply (PETScMatrix)	1	0.0010845	0.0010845
Apply (PETScVector)	3011	7.6334e-06	0.022984
Assemble cells	601	0.084259	50.64
Assemble rhs	600	0.084973	50.984
Build BoxMesh	1	0.020403	0.020403
Build sparsity	1	0.032	0.032
Compute SCOTCH graph re-ordering	3	0.0089569	0.026871
Compute connectivity 0-3	1	0.022979	0.022979
Compute connectivity 2-3	1	0.025353	0.025353
Compute entities dim = 2	1	0.21921	0.21921
Delete sparsity	1	2.2e-06	2.2e-06
Init dof vector	4	0.01166	0.046641
Init dofmap	3	0.2462	0.73859
Init dofmap from UFC dofmap	3	0.032292	0.096875
Init tensor	1	0.0031651	0.0031651
Merge step	600	0.0013864	0.83186
Number distributed mesh entities	3	8.3333e-07	2.5e-06
ODE step	1200	0.10137	121.64
PDE Step	600	0.092932	55.759
PETSc Krylov solver	600	0.0077343	4.6406
PointIntegralSolver::apply	1200	3.9632e-05	0.047558
PointIntegralSolver::step	1200	0.10012	120.14
SCOTCH: call SCOTCH_graphBuild	3	4.8433e-05	0.0001453
SCOTCH: call SCOTCH_graphOrder	3	0.0062986	0.018896
SplittingSolver: setup	1	2.1359	2.1359
SplittingSolver: solve (and store)	1	179.88	179.88

Listing A.1: Detailed insights in the wall time used by the different steps in the cbcbeat solver for a serial run using $\Delta x = 0.2$ mm, $\Delta t = 0.05$ ms, and $T = 30$s. The table outputted to the terminal shows the number of times a step is applied, the average wall time, and the total wall time contribution from each step.

[MPI_AVG] Summary of timings	reps	wall avg	wall tot
Apply (PETScMatrix)	1	0.0046709	0.0046709
Apply (PETScVector)	3011	0.00091881	2.7665
Assemble cells	601	0.015005	9.0183
Assemble rhs	600	0.018585	11.151
Build BoxMesh	1	0.50425	0.50425
Build LocalMeshData	1	0.033457	0.033457
Build distributed mesh	1	0.4468	0.4468
Build local part of dist. mesh	1	0.011842	0.011842
Build sparsity	1	0.017188	0.017188
Compute SCOTCH graph re-ordering	3	0.0012951	0.0038855
Compute connectivity 0-3	1	0.0041027	0.0041027
Compute connectivity 2-3	1	0.0046782	0.0046782
Compute entities dim = 2	1	0.042974	0.042974
Compute graph partition (SCOTCH)	1	0.23685	0.23685
Compt local part of mesh dual graph	1	0.032676	0.032676
Compt mesh entity ownership	1	0.014948	0.014948
Compt non-local p.o mesh dual graph	1	0.0099448	0.0099448
Delete sparsity	1	2.2833e-06	2.2833e-06
Distribute cells	1	0.019804	0.019804
Distribute mesh (cells and vertices)	1	0.035209	0.035209
Distribute vertices	1	0.0061505	0.0061505
Extract part boundaries SCOTCH graph	1	0.000813	0.000813
Get SCOTCH graph data	1	2.8833e-06	2.8833e-06
Init dof vector	4	0.0059096	0.023638
Init dofmap	3	0.046001	0.138
Init dofmap from UFC dofmap	3	0.0062377	0.018713
Init tensor	1	0.0011349	0.0011349
Merge step	600	0.0017218	1.0331
Number distributed mesh entities	4	0.015946	0.063786
Number mesh entities dist. mesh	1	0.05303	0.05303
ODE step	1200	0.020182	24.218
PDE Step	600	0.026665	15.999
PETSc Krylov solver	600	0.0078543	4.7126
PointIntegralSolver::apply	1200	0.0010946	1.3136
PointIntegralSolver::step	1200	0.018909	22.691
SplittingSolver: setup	1	1.5513	1.5513
SplittingSolver: solve (and store)	1	42.756	42.756

Listing A.2: Detailed insights in the wall time used by the different steps in the cbcbeat solver for a parallel run with 6 MPI threads using $\Delta x = 0.2$ mm, $\Delta t = 0.05$ ms, and $T = 30$s. The table outputted to the terminal shows the number of times a step is applied, the average wall time, and the total wall time contribution from each step.

A.2 Detailed Timings of openCARP

```
Reading reference mesh in 3.785443 sec.
Submesh Extraction done in 0.038463 sec.
Partitioning done in 0.000026 sec.
Redistributing done in 0.038989 sec.
Canonical numbering done in 0.046382 sec.
PETSc numbering done in 0.016009 sec.

All done in 3.932172 sec.

Computed parabolic stiffness matrix in 0.167 seconds.
Computed parabolic mass matrix in 0.119 seconds.
Initializing parabolic solver in 0.00089 seconds.

Timings of individual physics:
------------------------------

Electrics:
  Init:    0.44 sec
  Compute: 12.85 sec
  Output:  0.05 sec

Myocard Ionics:
  Init:    0.08 sec
  Compute: 6.23 sec
  Output:  0.00 sec

Total elapsed time: 23.811225652694702
```

Listing A.3: Detailed insights in the wall time used by the different steps in the openCARP solver using $\Delta x = 0.2$ mm, $\Delta t = 0.05$ ms, and $T = 30$s. The information is displayed as it is outputted to terminal with small modifications.

```
Reading reference mesh in 3.167930 sec.
Submesh Extraction done in 0.012242 sec.
Partitioning done in 0.014515 sec.
Redistributing done in 0.016080 sec.
Canonical numbering done in 0.018074 sec.
PETSc numbering done in 0.009960 sec.

All done in 3.250734 sec.

Computed parabolic stiffness matrix in 0.056 seconds.
Computed parabolic mass matrix in 0.043 seconds.
Initializing parabolic solver in 0.00095 seconds.

Timings of individual physics:
------------------------------

Electrics:
  Init:    0.19 sec
  Compute: 11.31 sec
  Output:  0.07 sec

Myocard Ionics:
  Init:    0.08 sec
  Compute: 1.74 sec
  Output:  0.00 sec


    *** Destroying physics ***

Destroying Electrics ..
Total elapsed time: 16.885425090789795
```

Listing A.4: Detailed insights in the wall time used by the different steps in the openCARP solver for optimal parallel execution, which for openCARP is 4 threads given $\Delta x = 0.2$ mm, $\Delta t = 0.05$ ms, and $T = 30$s. The information is displayed as it is outputted to terminal with small modifications.

Bibliography

Abhyankar, Shrirang, Jed Brown, Emil M Constantinescu, Debojyoti Ghosh, Barry F Smith, and Hong Zhang (2018). "PETSc/TS: A Modern Scalable ODE/DAE Solver Library". In: *arXiv preprint arXiv:1806.01437*.

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

Alnæs, Martin S., Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells (2015). "The FEniCS Project Version 1.5". In: *Archive of Numerical Software* 3.100. DOI: 10.11588/ans.2015.100.20553.

Beeler, Go W and H Reuter (1977). "Reconstruction of the action potential of ventricular myocardial fibres". In: *The Journal of physiology* 268.1, pp. 177–210.

Berdowski, Jocelyn, Robert A Berg, Jan GP Tijssen, and Rudolph W Koster (2010). "Global incidences of out-of-hospital cardiac arrest and survival rates: systematic review of 67 prospective studies". In: *Resuscitation* 81.11, pp. 1479–1487.

CreativeCommons (2021a). *Attribution 4.0 International (CC BY 4.0)*. URL: https://creativecommons.org/licenses/by/4.0/.

— (2021b). *Attribution-ShareAlike 3.0*. URL: https://creativecommons.org/licenses/by-sa/3.0/.

— (2021c). *Public Domain*. URL: https://creativecommons.org/share-your-work/public-domain/.

Cuellar, Autumn A, Catherine M Lloyd, Poul F Nielsen, David P Bullivant, David P Nickerson, and Peter J Hunter (2003). "An overview of CellML 1.1, a biological model description language". In: *Simulation* 79.12, pp. 740–747.

Farrell, Patrick E, David A Ham, Simon W Funke, and Marie E Rognes (2013). "Automated derivation of the adjoint of high-level transient finite element programs". In: *SIAM Journal on Scientific Computing* 35.4, pp. C369–C393.

Faul, Anita C (2018). "Linear Systems". In: *A Concise introduction to numerical analysis*. CRC Press. Chap. 2.

Hatton, L. and A. Roberts (1994). "How accurate is scientific software?" In: *IEEE Transactions on Software Engineering* 20.10, pp. 785–797. DOI: 10.1109/32.328993.

Henriquez, Craig S and Wenjun Ying (2021). "The bidomain model of cardiac tissue: from microscale to macroscale". In: *Cardiac Bioelectric Therapy*. Springer, pp. 211–223.

Hodgkin, Alan L and Andrew F Huxley (1952). "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of physiology* 117.4, pp. 500–544.

Hodgkin, Alan L, Andrew F Huxley, and Bernard Katz (1952). "Measurement of current-voltage relations in the membrane of the giant axon of Loligo". In: *The Journal of physiology* 116.4, pp. 424–448.

Katz, Arnold M (2011a). *Physiology of the heart*. 5th ed. Wolters Kluwer Health/Lippincott Williams & Wilkins Health. ISBN: 1-4511-4912-3.

— (2011b). "Structure of the Heart and Cardiac Muscle". In: *Physiology of the heart*. 5th ed. Wolters Kluwer Health/Lippincott Williams & Wilkins Health. Chap. 1, pp. 4–33. ISBN: 1-4511-4912-3.

— (2011c). "The Cardiac Action Potential". In: *Physiology of the heart*. 5th ed. Wolters Kluwer Health/Lippincott Williams & Wilkins Health. Chap. 14, pp. 370–401. ISBN: 1-4511-4912-3.

Keener, James P and James Sneyd (1998a). "Calcium Dynamics". In: *Mathematical physiology*. Second edition. Vol. 1. Springer. Chap. 7.

— (1998b). "Cellular Homeostasis". In: *Mathematical physiology*. Second edition. Vol. 1. Springer. Chap. 2.

— (1998c). “Excitability”. In: *Mathematical physiology*. Second edition. Vol. 1. Springer. Chap. 5.

— (1998d). *Mathematical physiology*. Second edition. Vol. 1. Springer.

— (1998e). “The Heart”. In: *Mathematical physiology*. Second edition. Vol. 1. Springer. Chap. 12.

Langtangen, Hans Petter and Anders Logg (2016). *Solving PDEs in python: the FEniCS tutorial I*. Springer Nature.

Logg, A., K.-A. Mardal, and G. N. Wells et al. (2012). “Automated Solution of Differential Equations by the Finite Element Method”. In:

Logg, Anders and Garth N. Wells (2010). “DOLFIN: Automated Finite Element Computing”. In: *ACM Transactions on Mathematical Software* 37.2. DOI: 10.1145/1731022.1731030.

Luo, Ching-hsing and Yoram Rudy (1994). “A dynamic model of the cardiac ventricular action potential. I. Simulations of ionic currents and concentration changes.” In: *Circulation research* 74.6, pp. 1071–1096.

Marsh, Megan E, Saeed Torabi Ziaratgahi, and Raymond J Spiteri (2012). “The secrets to the success of the Rush–Larsen method and its generalizations”. In: *IEEE transactions on biomedical engineering* 59.9, pp. 2506–2515.

McAllister, Ro E, D Noble, and RW Tsien (1975). “Reconstruction of the electrical activity of cardiac Purkinje fibres.” In: *The Journal of physiology* 251.1, pp. 1–59.

Merkel, Dirk (2014). “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239, p. 2.

Mitusch, Sebastian K, Simon W Funke, and Jørgen S Dokken (2019). “dolfin-adjoint 2018.1: automated adjoints for FEniCS and Firedrake”. In: *Journal of Open Source Software* 4.38, p. 1292.

Niederer, Steven A, Eric Kerfoot, Alan P Benson, Miguel O Bernabeu, Olivier Bernus, Chris Bradley, Elizabeth M Cherry, Richard Clayton, Flavio H Fenton, Alan Garny, et al. (2011). “Verification of cardiac tissue electrophysiology simulators using an N-version benchmark”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 369.1954, pp. 4331–4351.

Plank, Gernot, Axel Loewe, Aurel Neic, Christoph Augustin, Yung-Lin Huang, Matthias A.F. Gsell, Jorge Sanchez Elias Karabelas Mark Nothstein, Anton J Prassl, Gunnar Seemann, and Edward J Vigmond (2021a). *openCARP - Cardiac Electrophysiology Simulator*. URL: https://opencarp.org/.

— (2021b). "The openCARP Simulation Environment for Cardiac Electrophysiology". In: under review, bioRxiv preprint available. DOI: 10.1101/2021.03.01.433036.

Qu, Zhilin and Alan Garfinkel (1999). "An advanced algorithm for solving partial differential equation in cardiac conduction". In: *IEEE Transactions on Biomedical Engineering* 46.9, pp. 1166–1168.

Rognes, Marie, Patrick Farrell, Simon Funke, Johan Hake, and Mary Maleckar (May 2017). "cbcbeat: an adjoint-enabled framework for computational cardiac electrophysiology". In: *The Journal of Open Source Software* 2. DOI: 10.21105/joss.00224.

Rognes, ME, J Hake, PE Farrell, and SM Funke (2017). *Docs - cbcbeat: an adjoint-enabled framework for computational cardiac electrophysiology*. URL: https://cbcbeat.readthedocs.io/en/latest/.

Rush, Stanley and Hugh Larsen (1978). "A practical algorithm for solving dynamic membrane equations". In: *IEEE Transactions on Biomedical Engineering* 4, pp. 389–392.

Sundnes, Joakim, Glenn Terje Lines, Xing Cai, Bjørn Frederik Nielsen, Kent-Andre Mardal, and Aslak Tveito (2007a). "Computational Models". In: *Computing the electrical activity in the heart*. Vol. 1. Springer Science & Business Media. Chap. 3.

— (2007b). *Computing the electrical activity in the heart*. Vol. 1. Springer Science & Business Media.

— (2007c). "Solving Linear Systems". In: *Computing the electrical activity in the heart*. Vol. 1. Springer Science & Business Media. Chap. 4.

Ten Tusscher, Kirsten HWJ and Alexander V Panfilov (2006). "Alternans and spiral breakup in a human ventricular tissue model". In: *American Journal of Physiology-Heart and Circulatory Physiology* 291.3, H1088–H1100.

The HDF Group (2021). *Hierarchical data format version 5*. URL: http://www.hdfgroup.org/HDF5.

TheRoyalSociety (2021). *Copyright and intellectual property.* URL: https://royalsociety.org/journals/ethics-policies/plagiarism-copyright-ip/.

Tung, Leslie (1978). "A bi-domain model for describing ischemic myocardial dc potentials." PhD thesis. Massachusetts Institute of Technology.

Tusscher, Kirsten HWJ ten, Denis Noble, Peter-John Noble, and Alexander V Panfilov (2004). "A model for human ventricular tissue". In: *American Journal of Physiology-Heart and Circulatory Physiology* 286.4, H1573–H1589.

Virani, Salim S, Alvaro Alonso, Hugo J Aparicio, Emelia J Benjamin, Marcio S Bittencourt, Clifton W Callaway, April P Carson, Alanna M Chamberlain, Susan Cheng, Francesca N Delling, et al. (2021). "Heart disease and stroke statistics—2021 update: a report from the American Heart Association". In: *Circulation* 143.8, e254–e743.

Wang, Zhiguo, Bernard Fermini, and Stanley Nattel (1994). "Rapid and slow components of delayed rectifier current in human atrial myocytes". In: *Cardiovascular research* 28.10, pp. 1540–1546.

Xu, Haodong, Weinong Guo, and Jeanne M Nerbonne (1999). "Four kinetically distinct depolarization-activated K+ currents in adult mouse ventricular myocytes". In: *The Journal of general physiology* 113.5, pp. 661–678.

Yanagihara, Kaoru, Akinori Noma, and Hiroshi Irisawa (1980). "Reconstruction of sino-atrial node pacemaker potential based on the voltage clamp experiments". In: *The Japanese journal of physiology* 30.6, pp. 841–857.

Zipes, Douglas P and Hein JJ Wellens (2000). "Sudden cardiac death". In: *Professor Hein JJ Wellens*, pp. 621–645.