High-Performance Computing for Computational Biology of the Heart

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy

by

Ross M^{c}Farlane

October 2010

To Meg, Arthur & Ernest

"Disciplining yourself to do what you know is right and important, although difficult, is the highroad to pride, self-esteem, and personal satisfaction."

— Margaret "The Iron Lady" Thatcher

ABSTRACT

This thesis describes the development of Beatbox — a simulation environment for computational biology of the heart. Beatbox aims to provide an adaptable, approachable simulation tool and an extensible framework with which High Performance Computing may be harnessed by researchers.

Beatbox is built upon the QUI software package, which is studied in Chapter 2. The chapter discusses QUI's functionality and common patterns of use, and describes its underlying software architecture, in particular its extensibility through the addition of new software modules called 'devices'. The chapter summarises good practice for device developers in the *Laws of Devices*.

Chapter 3 discusses the parallel architecture of Beatbox and its implementation for distributed memory clusters. The chapter discusses strategies for domain decomposition, halo swapping and introduces an efficient method for exchange of data with diagonal neighbours called *Magic Corners*. The development of Beatbox's parallel Input/Output facilities is detailed, and its impact on scaling performance discussed. The chapter discusses the way in which parallelism can be hidden from the user, even while permitting the runtime execution user-defined functions. The chapter goes on to show how QUI's extensibility can be continued in a parallel environment by providing *implicit parallelism* for devices and defining *Laws of Parallel Devices* to guide third-party developers. Beatbox's parallel performance is evaluated and discussed.

Chapter 4 describes the extension of Beatbox to simulate anatomically realistic tissue geometry. Representation of irregular geometries is described, along with associated user controls. A technique to compute no-flux boundary conditions on irregular boundaries is introduced. The *Laws of Devices* are further developed to include irregular geometries. Finally, parallel performance of anatomically realistic meshes is evaluated.

ACKNOWLEDGEMENTS

I would like to acknowledge the contribution of my primary supervisor, Dr Irina Biktasheva, to this work.

I extend my heartfelt thanks to my second supervisors Professors Trevor Bench-Capon and Michael Fisher, whose input and support has been most helpful. This project would not have been possible without the input of Professor Vadim Biktashev, whose insight and clarity of thought have been invaluable. I am grateful to Dr Flavio Fenton and colleagues for the use of their rabbit mesh.

I am hugely indebted to the technical staff of the Computer Science department at the University of Liverpool — in particular Kenneth Chan, Dave Shield and Phil Jimmieson — who have consistently gone beyond the call of duty to help. I salute you!

The development of Beatbox has been greatly assisted by Cliff Addision, Dave Love and Ian Smith, of the University of Liverpool's Computing Services Department.

Finally, to everyone who has supported, assisted, counselled and comforted me over the last three years, thank you.

TYPOGRAPHICAL CONVENTIONS

Text in this thesis uses the following typographical conventions:

Computer code, or values represented in software are shown in `fixed width font`.

```
10 PRINT "Large sections of code are shown with a light-grey background."
20 GOTO 10
```

File names are set in italic type, as in *thesis.tex*.

Names of software packages are set in sans serif font, as in Beatbox.

From time to time, concepts are identified by capital letters, as in 'Run function'.

Latin text is set in italic type, *ad nauseum*.

CONTENTS

Abstract **vii**

Acknowledgements **ix**

Typographical Conventions **xi**

1 Introduction **1**
1.1 Overview . . . 1
1.1.1 Chapter Outline . . . 2
1.2 Cardiology . . . 2
1.2.1 Mechanics . . . 2
1.2.2 The Cardiac Conduction System . . . 2
1.2.3 Wave Propagation in Excitable Media . . . 4
1.2.4 Cardiac Arrhythmias . . . 8
1.3 Cardiac Electrophysiology . . . 8
1.3.1 Ion Channels . . . 10
1.3.2 Ionic Currents . . . 10
1.3.3 Conduction . . . 11
1.4 *In Silico* Modelling . . . 11
1.4.1 Motivation . . . 11
1.4.2 Examples . . . 12
1.5 Modelling Overview . . . 12
1.6 Cell Models . . . 13
1.6.1 Physiologically Detailed Models . . . 13
1.6.2 Reduced Models . . . 14
1.7 Tissue Models . . . 14
1.7.1 Bidomain . . . 15
1.7.2 Monodomain . . . 15
1.7.3 Boundary Conditions . . . 15
1.7.4 Anatomically Realistic Geometry of the Heart . . . 15
1.8 Reconstructing ECG . . . 16
1.9 Computational Methods . . . 16
1.9.1 Space Discretisation . . . 16
1.9.2 Time Discretisation . . . 17
1.10 Software Implementation . . . 18

1.10.1 Shared Memory Parallelism . . . 18
1.10.2 Distributed Memory Parallelism . . . 19
1.11 State of the Art Software Packages . . . 21
1.11.1 Carp . . . 21
1.11.2 Continuity . . . 21
1.11.3 CHASTE . . . 21
1.11.4 QUI . . . 23
1.12 The Beatbox Project . . . 23
1.12.1 The Case for Extending QUI . . . 23
1.13 Thesis Outline . . . 24

2 The QUI Software Package 25
2.1 Overview . . . 25
2.1.1 Modular Structure . . . 25
2.1.2 Running Simulations with QUI . . . 26
2.1.3 Chapter Structure . . . 28
2.2 QUI Scripts . . . 28
2.2.1 The QUI Scripting Language . . . 28
2.2.2 Preprocessing . . . 29
2.2.3 Defining Arithmetic Variables with the `def` Command . . . 31
2.2.4 Defining the Simulation Medium with the `state` Command . . . 33
2.2.5 Device Calls . . . 33
2.3 QUI Devices . . . 35
2.3.1 Condition . . . 35
2.3.2 `Space` . . . 35
2.3.3 Window . . . 36
2.3.4 Parameter Stucture . . . 36
2.3.5 Behaviour . . . 36
2.3.6 The Run Function . . . 37
2.3.7 The Destroy Function . . . 37
2.3.8 Template Macros . . . 38
2.3.9 The Create Function . . . 38
2.3.10 Name . . . 38
2.4 Building the Simulation . . . 40
2.4.1 The `init()` Function . . . 40
2.4.2 QPP — The QUI Preprocessor . . . 40
2.4.3 Defining k_variables . . . 42
2.4.4 Initialising the `state` Module . . . 43
2.4.5 Creating the Ring of Devices . . . 44
2.5 Running the Simulation . . . 47
2.6 Ending the Simulation . . . 47
2.6.1 The stop Device . . . 47
2.7 The `euler` Device . . . 48
2.7.1 Creating `euler` . . . 49
2.7.2 Running Euler . . . 53
2.8 Right Hand Side Modules . . . 53
2.8.1 The RHS Parameter Structure . . . 53
2.8.2 `u` . . . 55
2.8.3 Dependent Parameters . . . 55

2.8.4 Creating an RHS Module . . . 56
2.8.5 Running an RHS Module . . . 58
2.9 Diffusion Devices . . . 59
2.10 Devices for Setting Boundary Conditions . . . 59
2.10.1 Dirichlet Boundary Devices . . . 60
2.10.2 Neumann Boundary Devices . . . 60
2.10.3 Toroidal Boundary Devices . . . 60
2.11 The `k_func` Device . . . 62
2.11.1 Overview . . . 62
2.11.2 The `k_func` Parameter Structure . . . 62
2.11.3 Creating `k_func` . . . 62
2.11.4 Running `k_func` . . . 64
2.12 Input/Output Devices . . . 65
2.12.1 The `record` Device . . . 66
2.12.2 The `ppmout` Device . . . 66
2.12.3 The `dump` and `load` Devices . . . 67
2.13 Extending QUI . . . 68
2.14 Summary . . . 68

3 Beatbox: Parallelisation of QUI using MPI 69
3.1 Introduction . . . 69
3.1.1 Design Aims . . . 69
3.1.2 MPI — The Message Passing Interface . . . 70
3.2 Domain Decomposition . . . 70
3.2.1 Assigning Processes on the Supergrid . . . 73
3.2.2 Defining Subdomains . . . 73
3.2.3 Constraining Device Spaces . . . 77
3.2.4 Computing Supergrid Dimensions . . . 81
3.3 Inter-Process Communication . . . 82
3.3.1 `haloSwap()` . . . 85
3.3.2 Device-Specific Communication . . . 88
3.4 Preparing Devices for Parallelism . . . 89
3.4.1 Swapping Halos . . . 89
3.4.2 Literate Macros . . . 89
3.5 Boundary Conditions . . . 90
3.6 The `k_func` Device . . . 90
3.6.1 Conditions for Safety . . . 90
3.6.2 Restricting `k_func` for Parallel Safety . . . 91
3.6.3 Parallel Implementation . . . 93
3.6.4 Limitations of the Parallelised `k_func` Device . . . 95
3.7 The `sample` Device . . . 95
3.7.1 Creating `sample` . . . 95
3.7.2 Running `sample` . . . 96
3.8 The `reduce` device . . . 96
3.8.1 Creating `reduce` . . . 96
3.8.2 Running `reduce` . . . 97
3.9 The `poincare` Device . . . 97
3.9.1 Creating `poincare` . . . 97
3.9.2 Running `Poincare` . . . 98

3.10 MPI Input/Output . . . 99
3.10.1 File Partitioning . . . 99
3.10.2 File Handling . . . 99
3.11 The `dump` and `load` Devices . . . 102
3.12 The `ctlpoint` Device . . . 102
3.12.1 Creating the `ctlpoint` Device . . . 102
3.12.2 Running `ctlpoint` . . . 103
3.13 The `ppmout` Device . . . 103
3.13.1 Sequences In MPI . . . 103
3.13.2 Creating `ppmout` . . . 104
3.13.3 Running `ppmout` . . . 104
3.14 Verification . . . 104
3.15 MPI Performance . . . 105
3.15.1 `euler` Performance . . . 108
3.15.2 `diff` Performance . . . 108
3.15.3 I/O Performance . . . 108
3.16 Extending Beatbox . . . 112
3.17 Summary . . . 113

4 Anatomically Realistic Tissue Geometry 115
4.1 Introduction . . . 115
4.2 Reaction-Diffusion Equations & Neumann Boundary Conditions . . . 115
4.2.1 Isotropic Diffusion . . . 115
4.2.2 Anisotropic Diffusion . . . 116
4.3 Software Implementation Overview . . . 118
4.3.1 Aims of the Software Implementation . . . 118
4.4 User Interface . . . 118
4.5 Data Structures . . . 119
4.5.1 `Geom` and MPI . . . 121
4.6 Geometry Files . . . 121
4.6.1 Reading Geometry Files . . . 122
4.7 Adapting `euler` for Anatomically Realistic Simulations . . . 123
4.8 Adapting `ppmout` for Anatomically Realistic Simulations . . . 123
4.9 Neumann Boundary Conditions in the Isotropic Medium . . . 124
4.9.1 Explicit Method . . . 124
4.9.2 Laplacian with Implicit Neumann Boundary Conditions . . . 127
4.9.3 Comparison of Boundary Conditions Methods . . . 129
4.9.4 Implementation of Laplacian with Implicit Neumann Conditions . . . 129
4.10 Anisotropic Diffusion . . . 129
4.10.1 Discretisation of Reaction Diffusion Equations for Anisotropic Diffusion . . . 130
4.11 Anisotropic Diffusion with Implicit Neumann Boundary Conditions . . . 130
4.11.1 Anisotropy in Two Dimensions . . . 132
4.11.2 Implementation . . . 132
4.11.3 Script Interface . . . 133
4.12 Verification . . . 134
4.13 Geometry Performance with MPI . . . 135
4.13.1 `euler` Scaling Performance . . . 137
4.13.2 `diff` Scaling Performance . . . 139
4.14 Extending Beatbox with Geometry . . . 139

4.15 Summary 146

5 Conclusions **147**
5.1 Parallelism 147
5.2 Parallel Performance 148
5.3 Anatomically Realistic Tissue Geometry 148
5.4 Usability 148
5.5 Software Construction 149
5.5.1 Extensibility 149
5.6 Further Work 150
5.6.1 Optimising Domain Decomposition 150
5.6.2 Optimising Parallel I/O 150
5.6.3 Bidomain Modelling 150
5.6.4 CellML Integration 150
5.6.5 Automatic Translation of MRI Meshes 151
5.6.6 Graphical User Interface 151

A Code Listings for Chapter 2 **153**

B QUI Preprocessor Macros **157**
B.1 Integer 157
B.1.1 Usage 157
B.1.2 Parameters 157
B.2 Long Integer 157
B.2.1 Usage 157
B.2.2 Parameters 157
B.3 Real Number 158
B.3.1 Usage 158
B.3.2 Parameters 158
B.4 String 158
B.4.1 Usage 158
B.4.2 Parameters 158
B.5 File 158
B.5.1 Usage 158
B.5.2 Parameters 158
B.6 Code String 159
B.6.1 Usage 159
B.6.2 Parameters 159
B.7 Code Block 159
B.7.1 Usage 160
B.7.2 Parameters 160

C Code Listings for Chapter 3 **161**

D MPI Performance Scripts **177**

E Code Listings for Chapter 4 **181**

F Geometry Verification & Performance Scripts **183**

G Beatbox Scripting Guide **195**
G.1 The Beatbox Scripting Language . . . 195
G.2 Preprocessing . . . 196
G.2.1 Skipping Comments . . . 196
G.2.2 Including Other Beatbox Scripts . . . 196
G.2.3 Calling System Commands . . . 197
G.2.4 Expanding String Macros . . . 197
G.3 Defining Arithmetic Variables with the `def` Command . . . 198
G.3.1 Expressions . . . 198
G.4 Defining the Simulation Medium with the `state` Command . . . 199
G.4.1 Anatomically Realistic Tissue Geometry . . . 200
G.5 Device Calls . . . 200
G.5.1 Assigning Device Parameters . . . 200
G.5.2 Generic Device Parameters . . . 201
G.6 The `k_func` Device . . . 202
G.6.1 Phase Distribution . . . 203

H Beatbox Device Reference **205**
H.1 `clock` . . . 205
H.1.1 Overview . . . 205
H.1.2 Parameters . . . 205
H.2 `ctlpoint` . . . 205
H.2.1 Overview . . . 205
H.2.2 Parameters . . . 205
H.3 `diff` . . . 205
H.3.1 Overview . . . 205
H.3.2 Parameters . . . 206
H.4 `dump` . . . 206
H.4.1 Overview . . . 206
H.4.2 Parameters . . . 206
H.5 `euler` . . . 206
H.5.1 Overview . . . 206
H.5.2 Parameters . . . 206
H.6 `k_func` . . . 207
H.6.1 Overview . . . 207
H.6.2 Parameters . . . 207
H.7 `load` . . . 207
H.7.1 Overview . . . 207
H.7.2 Parameters . . . 207
H.8 `poincare` . . . 207
H.8.1 Overview . . . 207
H.8.2 Parameters . . . 208
H.9 `ppmout` . . . 208
H.9.1 Overview . . . 208
H.9.2 Parameters . . . 209
H.10 `record` . . . 209
H.10.1 Overview . . . 209
H.10.2 Parameters . . . 209
H.11 `reduce` . . . 210

H.11.1 Overview . . . 210
H.11.2 Parameters . . . 210
H.12 `sample` . . . 210
H.12.1 Overview . . . 210
H.12.2 Parameters . . . 210
H.13 `stop` . . . 210
H.13.1 Overview . . . 210
H.13.2 Parameters . . . 210

I Beatbox Developer Guide **211**
I.1 Hello Beatbox! . . . 211
I.1.1 The Device Skeleton . . . 211
I.1.2 Registering our Device . . . 212
I.1.3 Building Beatbox with our Device . . . 213
I.1.4 Running our Device . . . 214
I.2 Anatomy of a Device . . . 215
I.2.1 The Parameter Structure . . . 215
I.2.2 The `Device` Datatype . . . 215
I.2.3 Device Function Templates . . . 215
I.2.4 Acessing `New` . . . 217
I.2.5 Reading Parameters . . . 218
I.2.6 The Laws of Devices . . . 220

Bibliography **221**

CHAPTER

ONE

INTRODUCTION

1.1 Overview

Despite over a century's study, the trigger mechanisms of cardiac arrhythmias are poorly understood. Modern experimental methods do not provide sufficient temporal and spacial resolution to trace the development of fibrillation in samples of cardiac tissue, not to mention the heart *in vivo*. Computer simulations of electrical activity in cardiac tissue offer increasingly detailed insight into these phenomena, providing a view of cellular-level activity on the scale of whole heart chambers (atria or ventricles). Already, advances in this field have led to developments in our understanding of *fibrillation* and *sudden cardiac death* and their impact is expected to increase significantly as we approach the ultimate goal of whole-heart modelling.

Computer simulations of the heart involve vast scale ranges, with activity at μm and μs scales being observed in several cm^3 of tissue over tens of seconds. Moreover, modern biophysically realistic models of cardiac muscle cells greatly increase the computational complexity of simulations. Simulating the heart's irregular geometry and anisotropy further increase the computational demands. In this context, the timely completion of simulations relies on modern High Performance Computing (HPC) hardware.

Use of HPC facilities, although essential, is limited by the ad-hoc manner of software development seen across the research community, where many small software development projects are undertaken in isolation, each supporting a particular short-to-medium-term aim. Code developed in these environments is often poorly structured and documented — hampering reuse within the lab — and is unlikely to be portable, preventing use on different platforms elsewhere. Results produced from such software are difficult to compare or repeat, limiting the scope of peer review. This approach to development excludes a great many members of the community for whom code-level involvement with simulations is neither desirable nor practical. At the opposite end of the spectrum, large, general purpose simulation tools may provide the necessary functionality, HPC support and portability, but intimidate users with complexity or unapproachable user interfaces. The disparity of cardiac simulation software at present is, arguably, impeding scientific progress.

This thesis describes the development of Beatbox — a simulation environment for computational biology of the heart. Beatbox aims to provide an adaptable, approachable simulation tool and an extensible framework with which High Performance Computing may be harnessed by researchers.

1.1.1 Chapter Outline

This chapter begins by introducing the physiology of the heart and the pathologies for which computer models offer new insights. We then motivate the computer modelling of the heart, and describe the mathematical and computational techniques used to do so. We then discuss the current state-of-the-art software packages. Finally, we describe the aims and methods of the Beatbox project and outline the remainder of the thesis.

1.2 Cardiology

The heart is central to the circulatory system, pumping deoxygenated blood from the body to the lungs and returning newly oxygenated blood from the lungs to the body, as illustrated in Figure 1.1. Its regular contractions can vary in frequency from 40–200 beats per minute to serve the needs of the body.

1.2.1 Mechanics

The heart is divided by a septum into halves, each of which contains two chambers, an atrium and a ventricle. The left side of the heart receives blood from the lungs via the pulmonary veins into the left atrium. At the same time, blood from the body is received via the vena cava into the right atrium. The atrio-ventricular valves between atria and ventricles are open to blood flow in this direction, and the ventricles become partially filled. When the ventricles are nearly full, both atria contract, ejecting blood into the ventricles, filling them to capacity. Atrio-ventricular valves then close to prevent bloodflow back into the atria. After a short delay, the ventricles contract, expelling blood through the aorta (left side) and pulmonary artery (right side).

Cardiac Tissue Structure

As stated by LeGrice [2001], "it is most appropriate to view the ventricular myocardium as a three-dimensional hierarchy of interconnecting muscle layers."

Myocytes are arranged as fibres, connected end-to-end by intercalated discs and transversely by smaller interdigitations. Individual myocytes are held together by a network of connective tissue called the *endomysium*. Tissue in the heart has a laminar structure; fibres lie in parallel in *sheets*, which are around 4 fibres thick [LeGrice, 2001] and are separated by layers of connective tissue called the *perimysium*. Sheets are stacked to form the tissue wall, which is ultimately encased in the *epimysium* [Katz, 2006]. The orientations of fibres and sheet planes vary throughout the tissue wall to form a helical pattern, spiralling out from the apex of the heart to the base.

1.2.2 The Cardiac Conduction System

A single heartbeat requires the contraction of around 10^{10} muscle cells [Sundnes et al., 2006]. The complex sequence of actions described in Section 1.2.1 is orchestrated by a wave of electrical excitation through the heart's conduction system. This electrical excitation is due to the change of voltage difference across each individual cell membrane and is called *action potential* (AP). During an action potential pulse, the voltage across a cell's membrane will change rapidly from its negative resting value to a positive, excited state. Figure 1.2 shows the time course of transmembrane voltage in a typical action potential pulse.

Unlike skeletal muscle, the beating of the heart is not directly controlled by the brain. Cardiac myocytes are *myogenic*, that is they are capable of contracting without 'instruction' from nerve

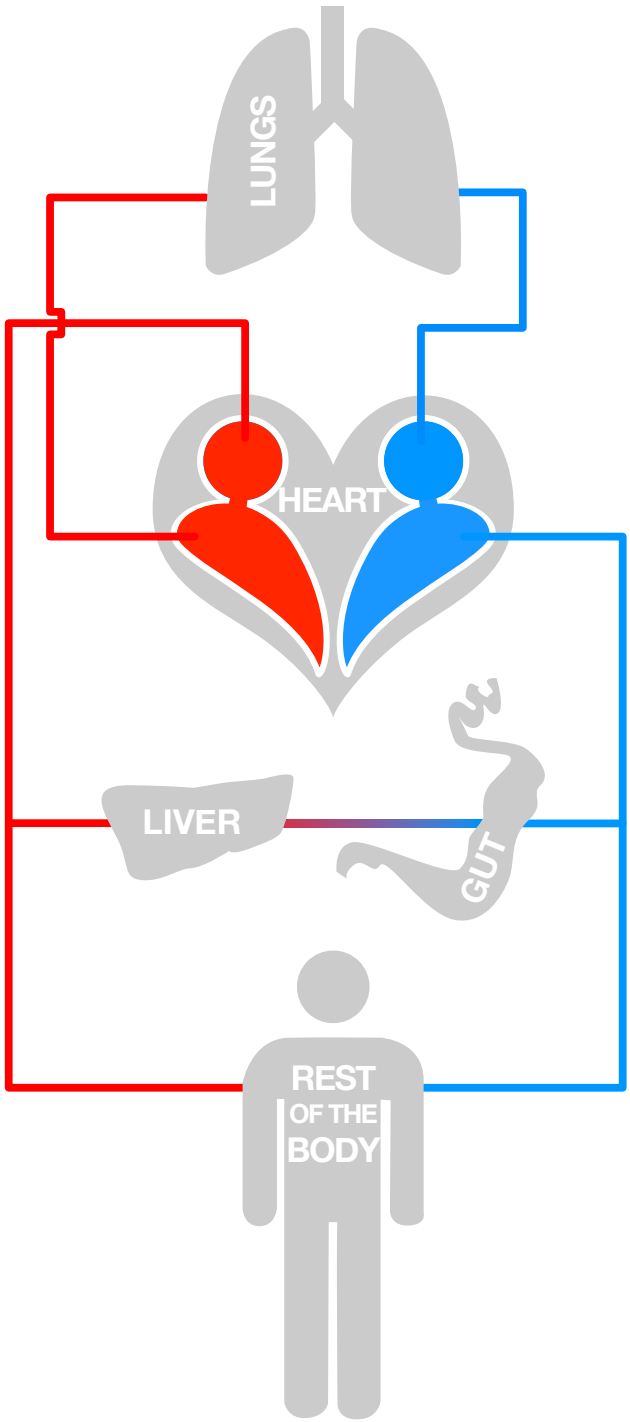


Figure 1.1: The Human Circulation System. Oxygenated bloodflow is shown in blue; deoxygenated bloodflow is shown in red. Adapted from Suntharesswaran [1998].

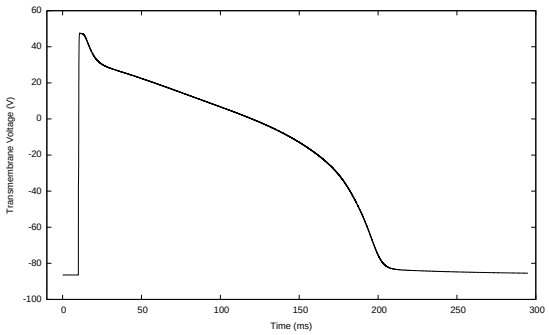


Figure 1.2: A typical, solitary action potential of a ventricular myocyte. Simulated with the Luo and Rudy [1994a] model of ventricular action potential using the QUI software package. $\Delta t = 0.01$, D = 0.03125, $\Delta x = 0.1$. The pulse is triggered by increasing the transmembrane voltage (V) by 50mV.

impulses. When a cardiac myocyte is excited above a threshold level, it generates an AP pulse during which the cell rapidly contracts and then slowly relaxes. During its action potential pulse, a myocyte is capable of exciting other nearby cells. Bundles of *autorhythmic* cells at various locations in the heart possess a chemical instability that causes them to trigger their own AP pulse with a certain frequency. A normal heartbeat begins with the activation of autorhythmic cells in the *sinoatrial node* (SAN), located in the right atrium, which acts as the heart's main pacemaker.

Figure 1.3 shows the cardiac conduction system and sequence of activation. The AP pulse from the SAN propagates through fibres and tissue, initiating atrial contraction as it passes. The atria are isolated from the ventricles by a 'skeleton' of unexcitable connective tissue. The only route for AP propagation between atria and ventricles is via the *atrio-ventricular node* (AVN). It is the slow conduction of the AVN that introduces the delay between atrial and ventricular contractions, described above. Excitation then travels through the Bundle of His, before dividing into left and right bundle branches and ultimately into the network of *Purkinje fibres* to excite the endocardial side of the ventricles. Excitation then spreads through the ventricular myocardium causing contraction.

The structure of cardiac tissue makes the conductivity of the myocardium strongly anisotropic. The end-to-end connection of myocytes makes conduction far greater along the muscle fibre. Conduction between adjacent fibres is reduced, and conduction between adjacent sheets is less still.

1.2.3 Wave Propagation in Excitable Media

The propagation of AP through cardiac tissue is due to the excitability of cardiac myocytes. AP propagates as one excited cell produces an over-threshold excitation for its neighbours, who become excited and in turn excite their neighbours. Following excitation, a cell will settle back down to rest. At any point in time, the electrochemical balance of a cell can be classified into three states [Wiener and Rosenbleuth, 1946]:

Rest At rest, cells have no effect on their neighbours. If neighboured by an excited cell, they can become excited.

Excitation Stimulus above the cell's threshold level will cause it to become excited. When

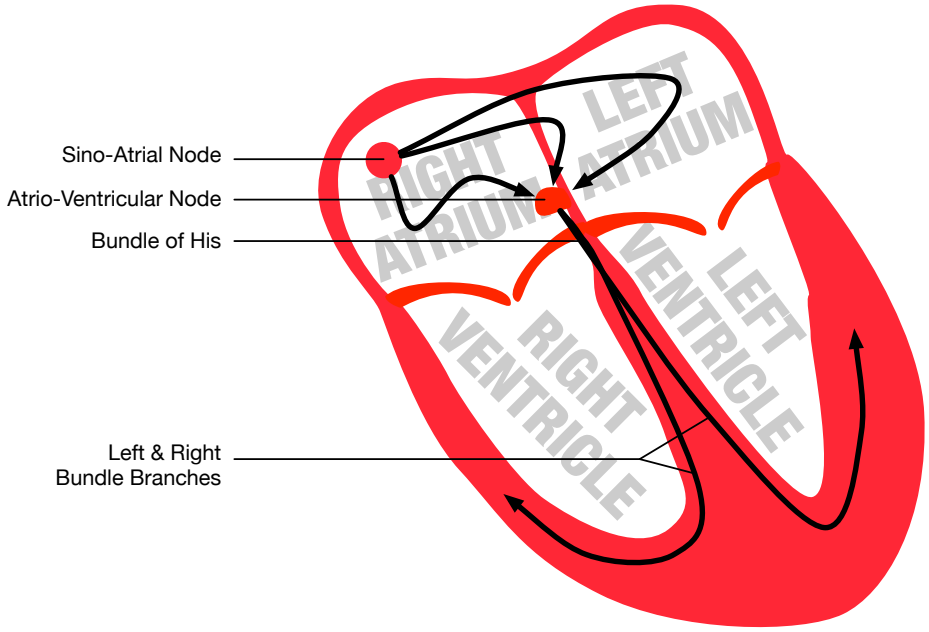


Figure 1.3: The cardiac conduction system.

excited, cells can excite their neighbours. The level of excitation is unaffected by neighbouring cells. Once excited, a cell will only remain excited for a short time, before entering refractoriness.

Refractoriness Having been excited, the cell slowly recovers towards rest. A cell in the refractory state cannot excite neighbouring cells, nor be excited by them. Typically the refractory phase will last hundreds of milliseconds.

Figure 1.4 shows the propagation of excitation along a 1-dimensional medium. In the example shown, refractoriness lasts twice as long as excitation. This gives a *refractory tail* of two cells behind the excited cell. As these cells cannot be excited again until they reach rest, they prevent the wave of propagation from moving backwards.

In the event that two waves of propagation, travelling in opposite directions should meet, the refractory tails of each wave prevent them from crossing over and they will instead annihilate each other. This is illustrated in Figure 1.5. Annihilation of opposing waves of propagation gives rise to the *Synchronisation Effect*. If the medium is stimulated at both ends with differing frequencies, the point of annihilation will drift towards the source of the slower stimulus, suppressing it. This is known as *overdrive suppression* [see, e.g. Dubin, 2003]. Consequently, in an excitable medium, the fastest stimulus will ultimately suppress all other stimuli and become the centre, or *focus*, of propagation for the medium. The synchronisation effect is shown in Figure 1.6. In the heart, the SAN is the fastest of the natural sources of excitation, and therefore dictates heart rate. In the event of SAN failure, or the SAN pulse being unable to reach some cells, a hierarchy of slower autorhythmic cells are able to take over.

In two-dimensional media, excitation propagates radially from the source. Like linear waves, such as sound waves, waves of propagation in excitable media are able to diffract around objects. In the heart, an 'object' takes the form of inexcitable cells; either those damaged by trauma, or cells in the refractory state. The ability of waves to diffract facilitates the generation of spiral,

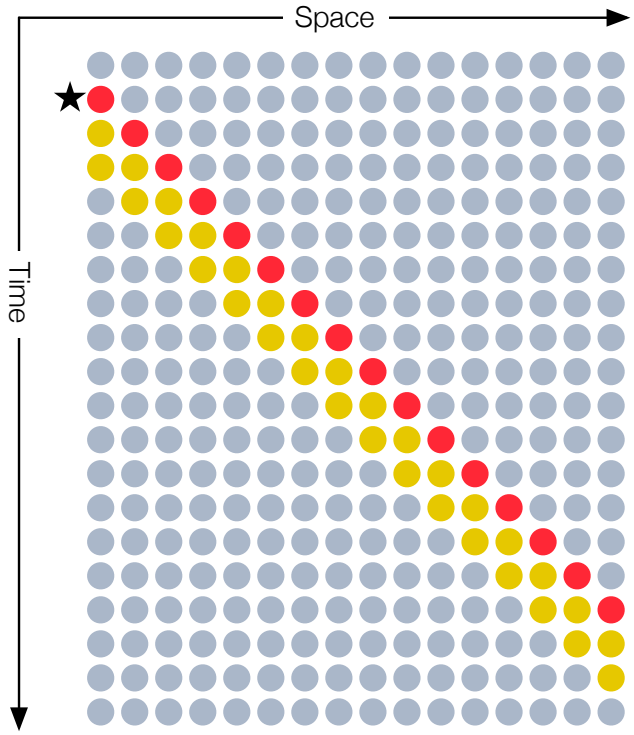


Figure 1.4: Conduction along a 1-dimensional medium. The medium is discrete in space and time, and cells have three discrete states: rest (blue), excitation (red) and refractoriness (yellow). A star marks the time step where the first cell is stimulated.

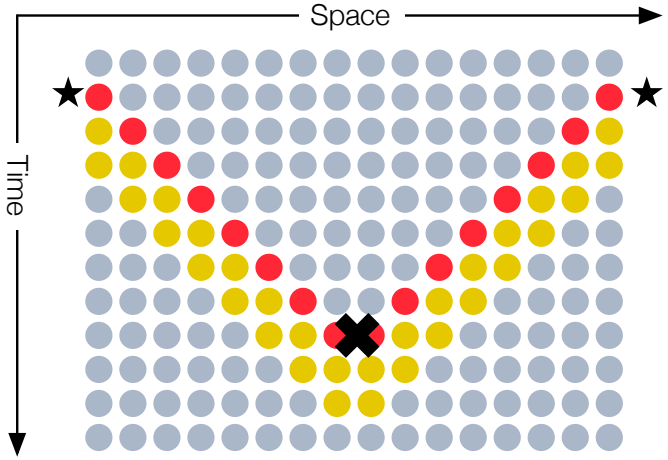


Figure 1.5: Annihilation of waves of excitation. The medium is discrete in space and time, and cells have three discrete states: rest (blue), excitation (red) and refractoriness (yellow). A cross marks the point of annihilation.

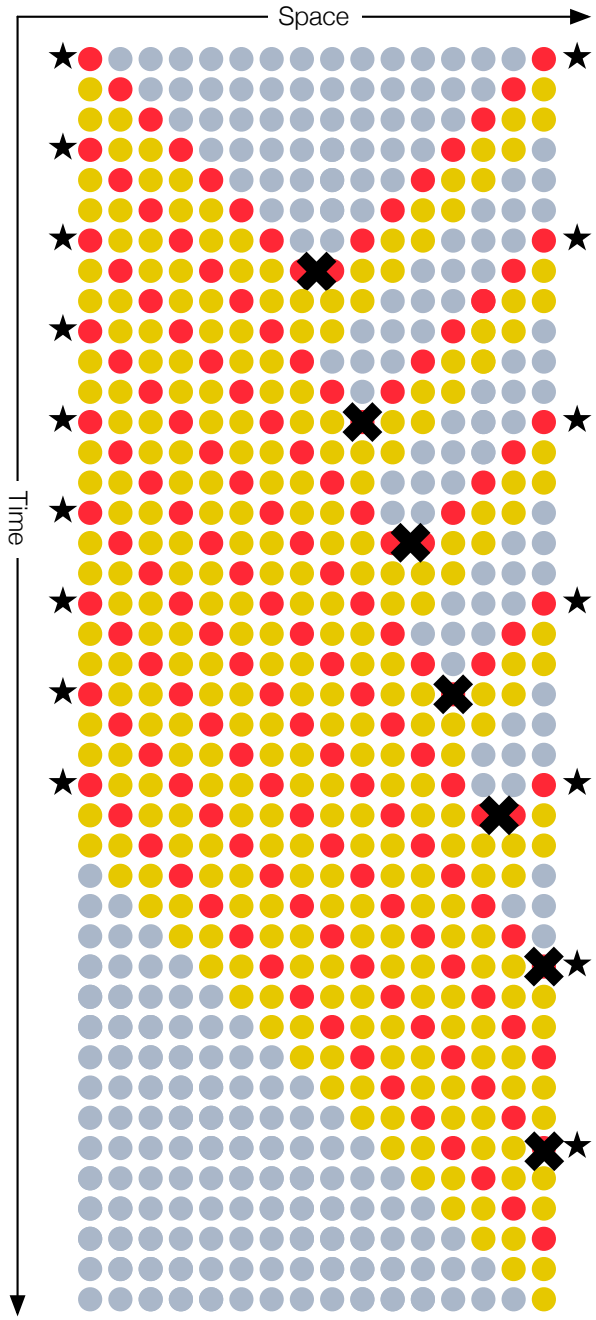


Figure 1.6: Overdrive suppression. A 1-dimensional medium is stimulated from both ends with differing frequencies. The medium is discrete in space and time, and cells have three discrete states: rest (blue), excitation (red) and refractoriness (yellow). Stars show the times of stimuli and crosses show the points of annihilation. Over time, the points of annihilation move closer to the slower stimulus, until excitation along the entire medium is dictated by the faster stimulus.

or reentrant, waves within the medium. This is a point of particular interest to cardiologists, as spiral waves form the basis of life-threatening arrhythmias [Mines, 1913, Allessie et al., 1973, Davidenko et al., 1992, Gray et al., 1998].

One cause of spiral wave formation in excitable media is shown in Figure 1.7. When a spiral wave forms in cardiac tissue, the waves of excitation that ripple from its centre act as a stimulus to the neighbouring tissue.

1.2.4 Cardiac Arrhythmias

Arrhythmias are malfunctions of cardiac pacing or conduction that cause a resting heart rate to fall outside of the normal range of 60–100 BPM. Some arrhythmias arise as a result of slowed pacemaker activity or additional conduction pathways between atria and ventricles. Of particular interest are *reentrant arrhythmias*.

According to Katz [2006], "Reentry occurs when a single impulse traveling through the heart gives rise to two or more propagated responses."

Reentrant arrhythmias can arise when plane-waves of excitation enter cyclic patterns, either along branching conduction pathways, or by refracting around areas of reduced or no conductivity. If the *pitch* of the reentrant wave is sufficiently tight, then the frequency of this stimulation may exceed that of the SAN, thus suppressing it and moving the focus of excitation.

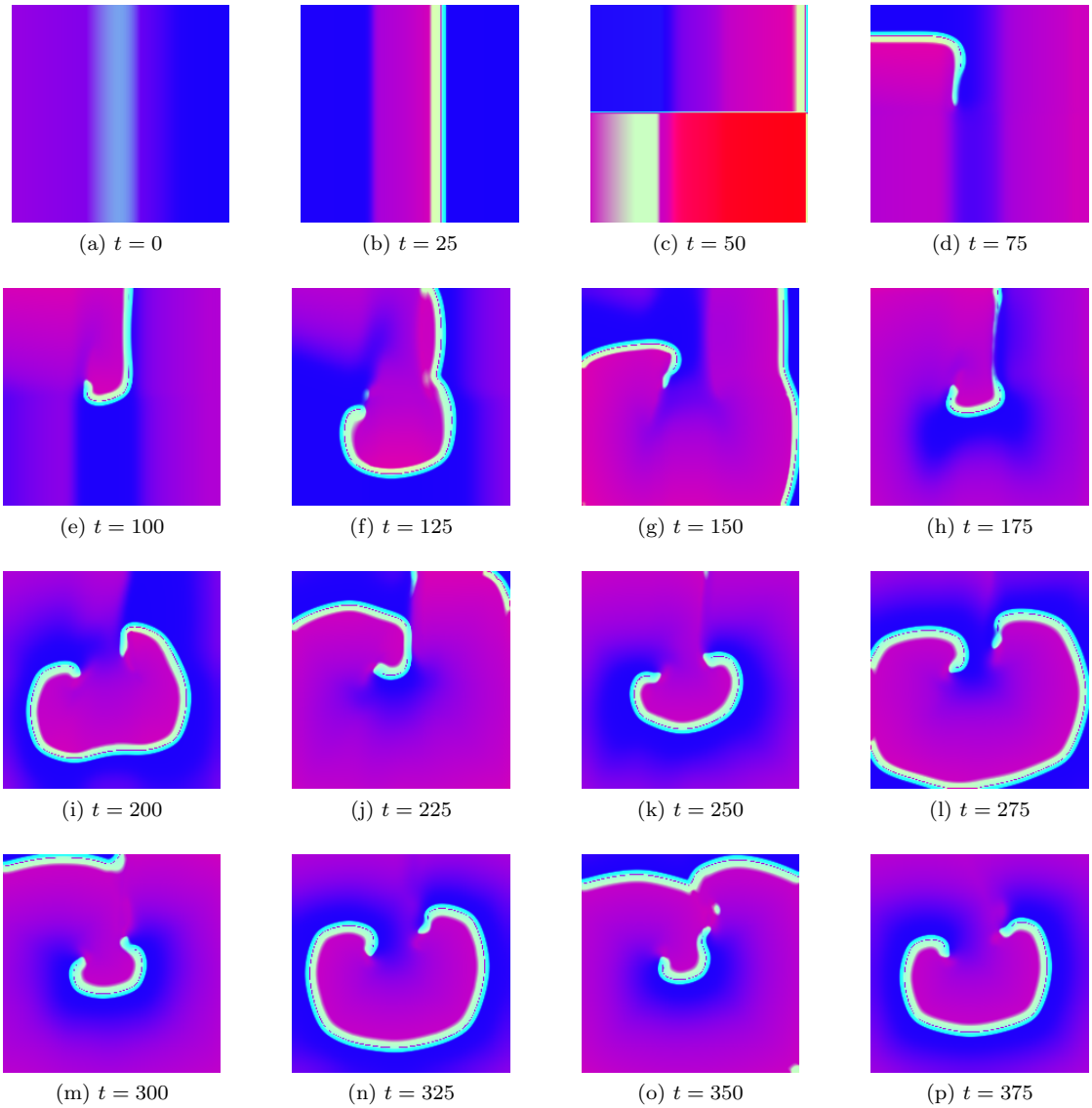
The breakup of reentrant waves can lead to *fibrillation* [Davidenko et al., 1992, Gray et al., 1998], where multiple reentrant waves create many, moving foci. Fibrillation is a chaotic pattern of activation that disorganises contraction, "where the fibrillating chamber resembles a bag of worms" [Katz, 2006]. When arising in the atria, fibrillation can prevent the atria from properly 'topping up' the ventricles prior to systole. It also increases the rate of ventricular contraction. In the ventricles, fibrillation will prevent cardiac output altogether, leading to death within minutes of onset.

The last 100 years have seen the development of hypothesis of cardiac reentry by Mines [1913] to experimental confirmation by Allessie et al. [1973], Davidenko et al. [1992], Gray et al. [1998]. In parallel, a lot of work was done on the electrophysiology of individual cardiac myocytes [Boyett et al., 1997]. Recent genetic studies have highlighted various mutations that predispose an individual to arrhythmias [e.g. Rudy, 2000, Veldkamp et al., 2000, Wilde and Bezzina, 2005, Watanabe et al., 2008]. However, the genesis of those arrhythmias remains poorly understood.

1.3 Cardiac Electrophysiology

The myocardium is comprised of millions of cardiac myocytes. While these vary in different regions of the heart, they share several properties regarding their electrical function. Each cell is comprised of a central nucleus, surrounded by cytoplasm and is ultimately contained by an insulating membrane. The membrane is permeable to water, oxygen and carbon dioxide molecules, but resists other molecules and ions under normal circumstances.

At rest, there exists a transmembrane potential difference caused by differing concentrations of different ions inside and outside of the cell. Without excitation, most species of myocyte will reach a resting state with a transmembrane potential of around -85mV. The potential difference between the cell and the surrounding 'bath' means that ions are effectively 'spring-loaded' and, given the opportunity, will be propelled down their *electrochemical gradient*. The electrochemical gradient describes the natural tendency of the ion to move in response to the imbalance of ions and/or potential difference across the membrane. An action potential entails a rapid depolarisation of the cell with respect to its surroundings, followed by a relatively long recovery.



(a) $t = 0$ (b) $t = 25$ (c) $t = 50$ (d) $t = 75$

(e) $t = 100$ (f) $t = 125$ (g) $t = 150$ (h) $t = 175$

(i) $t = 200$ (j) $t = 225$ (k) $t = 250$ (l) $t = 275$

(m) $t = 300$ (n) $t = 325$ (o) $t = 350$ (p) $t = 375$

Figure 1.7: Formation of a reentrant wave. Simulated using the Luo and Rudy [1994a] model of ventricular action potential using the CUI software package [reproduced from M^{c}Farlane, 2007]. Medium:400 $\times$ 400 points, $\Delta t = 0.01$, $\Delta x = 0.1$, D $= 0.03125$, stimuli: +100mV. The spiral wave is initiated using an S1-S2 stimulus protocol. The first (S1) stimulus initiates a plane wave moving from left to right, shown in (a) and (b). At (c), the second (S2) stimulus is applied to the lower half of the medium. In subsequent figures, the S1 wavefront can be seen to bend around the refractory cells created by the S2 stimulus.

Only when the cell has returned almost to its equilibrium conditions will it be able to respond to excitation and generate an action potential.

Following the movement of ions down their electrochemical gradient, the gradient begins to level out until the imbalance of ions has been redressed or the transmembrane potential zeroed. At this point the *reversal potential* is reached, beyond which, the direction of the ion's electrochemical gradient is reversed.

1.3.1 Ion Channels

The cell membrane is perforated by a number of proteins that facilitate the passage of certain ions into and out of the cell. These *ion channels* are selective for one or two species of ion, such as Ca^{2+} or K^{+}. The majority of the ion channels found in cardiomyocytes are *gated ion channels*, in that they are only 'open' to the passage of ions under certain conditions. The conditions required to open ion channels vary. Some may be triggered by an above threshold intra- or extra-cellular ion concentration, while others will be opened by variances in transmembrane voltage. Some will also be affected by the presence of 'messenger' molecules such as acetylcholine. When closing, some gates will simply 'swing' closed with the passage of time, some respond to particular conditions, and some close based on a combination of the two.

When an ion channel's conditions for opening are met, it allows the ions for which it is selective to pass down their *electrochemical gradient*, into or out of the cell. *Exchanger channels* allow the passage of outgoing ions in exchange for incoming ions of a different type. For example, taking in 3 Na^{+} ions in order to export 1 Ca^{2+} ion. In this exchange, the passage of Na^{+} ions down their electrochemical gradient powers the countertransport of Ca^{2+} ions.

The movement of ions down their electrochemical gradient is *passive*, as the ion channel expends no energy in moving ions, it merely opens the gate. *pumps* are a type of *active* ion channel that, when triggered, force ions to move up their electrochemical gradient, against the natural flow. For example, one pump transports Na^{+} from the cell while pumping K^{+} in.

1.3.2 Ionic Currents

While the behaviour of an individual ion channel is stochastic, the collective behaviour of a particular type of ion channel is more predictable. Taking a higher level view, electrophysiologists describe cells' behaviour as the product of several *ionic currents*, which to correspond to the collective behaviour of particular types of ion channel. Like ion channels, ionic currents are activated and deactivated by changes in cell state. Because an ionic current represents the collective behaviour of many of ion channels, its dynamics are a little more subtle. For example, gating happens gradually, rather than switching on and off, as ion channels do [Boyett et al., 1997]. The number of currents thought to exist in each myocyte changes frequently, and usually upwards, as new descriptions take an increasingly reductionist approach. The effect of ionic currents in general in defining the AP morphology is discussed below.

When the transmembrane voltage is raised to be more positive than a threshold potential ($\approx -65mV$), a positive inward Na^{+} current is opened. This causes the rapid depolarisation of the cell. Once the transmembrane potential is more positive than $-40mV$, Ca^{2+} ions enter the cell, causing it to contract. The influx of Ca^{2+} ions, countered by a fast K^{+} current helps to *plateau*, maintaining a transmembrane voltage of around $0mV$. Repolarisation occurs when further K^{+} currents activate and Ca^{2+} currents subside. This simplified description would seem to suggest that the AP ends with substantial changes in ion concentrations from rest, but such differences are redressed by pump and exchanger currents [Panfilov and Holden, 1997].

1.3.3 Conduction

Unlike copper cable in an electrical circuit, conduction in cardiac tissue does not come about from the movement of electrons. Rather, cells are able to pass ions to their neighbours through *intercalated discs* — groups of protrusions from the cell's ends that lock together like the fingers of clasping hands. Intercalated discs each contain many *connexons*. A connexon is similar to an ion channel, in that it perforates the membrane, providing a channel through which ions can pass. The connexon is then joined to one from a neighbouring cell, allowing ions to 'tunnel' through the connexon pair, across the intercellular space [Dubin, 2003].

At the peak of depolarisation, the influx of Na^+ ions exceeds the reversal potential of sodium currents and creates a potential difference across the membrane, which causes current flow through connexons to redress the balance. The electrochemical balance in recently excited cells will resist excitation. In the receiving cell, the net positive current raises the transmembrane potential above the threshold for its inward Na^+ current and triggers an action potential pulse.

1.4 *In Silico* Modelling

The field of Computational Biology of the Heart aims to produce an accurate mathematical model of the heart, capable of predicting its behaviour under varying chemical and physical conditions. Modelling the heart aims to augment the understanding gained through physical experimentation by offering greater spacial and temporal granularity with which to make observations.

1.4.1 Motivation

As Noble [2004] says, "Life-threatening arrhythmias depend on molecular and cellular mechanisms, but they are fatal because of their actions at the level of the whole organ." One of the primary motivations for *in silico* modelling of cardiac electrophysiology are differences in scale between the phenomena to be observed and the detail required to model them. Abramson et al. [2010] quantifies these differences as 10^9 spatially and 10^{15} temporally. The phenomena under study occur on the micron scale, with significant changes taking place in microseconds. Existing diagnostic and experimental methods are limited in their ability to capture this level of detail, particularly where high spacial and temporal resolutions are required simultaneously.

A number of techniques exist to measure the electrical activity in the heart. Macroscopic measurements of the cardiac conduction system may be made with an ECG, in which between 2 and 12 leads are placed in contact with the skin. Heavily relied upon by clinicians as a diagnostic tool, the ECG can be used in the identification of arrhythmias, but offers very little information as to their nature and no insight into their genesis.

Some studies, [e.g. Gray et al., 1997], have made use of a potentiometric dye, that changes colour with action potential, which can be applied to the exposed tissue. High-speed videography can then be used to record the changing colours on the surface of the tissue. While this technique offers high spatial resolution, it is limited by the sample rate of the video equipment. In addition, the use of such dyes requires that the heart be ex-vivo.

If it is possible to gain physical access to the living heart — for example where willing patients are undergoing surgery — it is possible to measure activity on the surface of the heart directly. Some human studies, such as Nash et al. [2006], Clayton et al. [2009] have surrounded the heart with a 'sock' of electrodes. This allows accurate measurements of surface action potentials to be made with high temporal resolution. The spacial resolution of the method is limited by the size and number of the electrodes.

Methods of this kind, as with all methods used in living patients are restricted to the epicardial surface, which limits their usefulness in studying the three-dimensional nature of action potential waves.

In animal tests, more invasive techniques may be used, such as the insertion of electrodes into the tissue. This technique makes it possible to measure mid-wall and endocardial action potentials by penetrating the tissue. As in the electrode sock, the spacial resolution of measurements is limited by the size and number of the electrodes. This technique is also extremely invasive and unsuitable for human patients.

Future improvements in the accuracy of experimental techniques will not lessen the need for *in silico* modelling. For example, while some arrhythmias of interest to researchers, such as Long QT syndrome or Brugada Syndrome are linked to specific genetic mutations [Clancy and Rudy, 2002, Veldkamp et al., 2000, Wilde and Bezzina, 2005], resultant arrhythmias occur without warning in otherwise healthy individuals and often result in death within minutes. In these cases, there is very little opportunity for medical intervention or study of any kind. *In silico* simulations afford researchers with the tools to recreate these kinds of otherwise obscure phenomena.

As computer modelling develops, there is an opportunity for it to play a role in solving so-called *inverse problems* [see, e.g. Gulrajani, 1998] faced by clinicians in diagnosing arrhythmias. Given *in vivo* measurements of pathological activity, from an ECG for example, we may one day be able to work backwards to find an underlying electrophysiological cause. This is a unique opportunity presented by *in silico* modelling.

1.4.2 Examples

Morgan [2009] used *in silico* simulations to assess and develop a low-voltage defibrillation method called Resonant Drift Pacing (RDT). The work compares termination of reentrant arrhythmias by RDT in anatomically realistic models of cardiac tissue. He describes a means of measuring excitation at a point in the tissue to inform the frequency and timing of an applied stimulus, which is able to terminate the turbulent scrollwaves of excitation present in fibrillation by driving them towards an inexcitable boundary. *in silico* models allowed Morgan to incite fibrillation in varying conditions and study the development of intramural scrollwaves by analysing both their fronts and *filaments*.

Butters et al. [2010] describes a simulation study used to link a known arrhythmogenic genetic mutation to dysfunction of the SAN, known as Sick Sinus Syndrome. They modelled the effects of the mutations on single cells, which were then integrated into a two-dimensional model of the rabbit right atrium. Their tissue model describes the varying cell types and conductivity throughout the tissue. Their simulations showed that "[...] the mutations not only slow down pacemaking, but also slow down the conduction across the SAN-atrium, leading to a possible SAN conduction exit block or sinus arrest, the major features of [Sick Sinus Syndrome]." *in silico* experimentation allowed the authors to develop hypotheses in great detail, before confirming them in isolated tissue experiments.

1.5 Modelling Overview

Models of the heart are comprised of several, modular components: models of cellular electrophysiology; a tissue model that describes the interconnection of those cells and the manner in which excitation will spread through the medium; and geometric models that give the tissue shape and structure, possibly including the anatomies of conductive systems. Detail at each of

these interacting scale levels is equally important; only in concert can they deliver the necessary realism.

Typically, a model of the individual cardiac myocyte serves as the starting point. As described by Noble [2002],

> There has been considerable debate over the best strategy for biological simulation, whether it should be "bottom-up", "top-down" or some combination of the two [...] The consensus is that it should be "middle out," meaning that we start modelling at the level(s) at which there are rich biological data and then reach up and down to other levels. In the case of the heart, we have benefited from the fact that, in addition to the data-rich cellular level, there has also been data-rich modelling of the 3D geometry of the whole organ [...]

1.6 Cell Models

Models of cardiac cells vary in both physiological detail and computational complexity. Broadly, there exist two approaches; discrete-state Markov-based models and those using systems of ordinary differential equations (ODEs), following the Hodgkin-Huxley formalism. A comparison of human ventricular cell models, including the Iyer et al. [2004] Markov-based model is given by Tusscher et al. [2006]. In addition, some cell models attempt to capture the stochasticity of ion channels. Here, we discuss only deterministic models following the Hodgkin-Huxley formalism.

1.6.1 Physiologically Detailed Models

Physiologically detailed models aim to reproduce the behaviour of cardiac cells by modelling their internal function based on the kinetics of the ion channels. Models of this kind follow the Hodgkin and Huxley formalism described in their Nobel prize-winning study of the squid giant axon [Hodgkin and Huxley, 1952]. In general, these models take the form

$$C_m \frac{\mathrm{d}V_m}{\mathrm{d}t} = -\sum_S I_S \tag{1.1}$$

where C_m is the membrane capacitance, V_m is the transmembrane voltage and I_S is the current carrying the S ion. The kinetics of ion channels is defined by

$$I_S = \bar{G}_S x_S^1 x_S^2 \dots x_S^n (V_m - V_S), \tag{1.2}$$

where $\bar{G}$ and V_S are the maximum conductance and reversal potential of I_S, respectively. x_S are gating variables that describe the openness of the channel. x_S will vary in response to changes in voltage, and tend towards a steady state, $x_{S\infty}$, after time τ_S, as defined by

$$\frac{\mathrm{d}x_S}{\mathrm{d}t} = \frac{x_{S\infty}(V_m) - x_S}{\tau_S(V_m)}. \tag{1.3}$$

The first adaptation of the Hodgkin-Huxley model for cardiac cells was made by Noble [1962] to describe cells of the Purkinje fibres. Both Hodgkin-Huxley and Noble's original systems had four ODEs, describing transmembrane voltage and three main ionic currents (Na^+, K^+, and a so-called 'leakage' current). The development of experimental facilities has allowed models to describe an increasing number of ionic currents, such as the introduction of the slow inward calcium current by Beeler and Reuter [1977], or active transports such as pump and exchanger currents. Further development has produced models that describe the intracellular concentrations of key

ionic species, [e.g. Courtemanche et al., 1998, Luo and Rudy, 1994a]. These models add a second equation of the form

$$\frac{\mathrm{d}[S]}{\mathrm{d}t} = \sum_k I_S^k(V_m, S), \tag{1.4}$$

where $[S]$ is the intracellular concentration of ionic species S, which is affected by k ionic transports and I_S^k is the current of ions of species S travelling through ionic transport k.

The Luo and Rudy [1994a] model adds further detail describing the intracellular buffering of Ca^{2+} and mechanism for its release into the myoplasm.

1.6.2 Reduced Models

Simplifications of physiologically accurate models have been used to emphasise particular aspects of the model's behaviour, e.g. dissipation of excitation wavefronts [Biktashev, 2002]; or, most often, to reduce the computational complexity of the model while retaining its qualitative features.

Simple, two-variable, models provide a phenomenological caricature of the cardiac action potential, whose behaviour can be easily plotted in two dimensions as a phase plane. The seminal model developed, independently, by FitzHugh [1961] and Nagumo, uses two dynamic variables, u and v, where u is an approximation of the transmembrane voltage and v represents the currents. It should be noted, however, that FitzHugh's aim was to caricature the cardiac action potential, not to simplify its computation.

$$\frac{\partial u}{\partial t} = f(u, v), \frac{\partial v}{\partial t} = g(u, v). \tag{1.5}$$

The computational complexity of physiologically detailed models, in particular second generation models, can present some difficulties when running large-scale simulations. Rather than rely upon the caricatures of simplified cell models, much work has been undertaken to reduce the detail, and thus complexity of physiologically detailed models, while retaining their phenomenological features. The reduction process may involve removing or unifying several ionic currents to provide a model informed but not burdened by physiological detail. Examples of reduced models are those by Barkley [1991], Fenton and Karma [1998], Bernus et al. [2002], Tusscher and Panfilov [2006].

1.7 Tissue Models

It is tempting to view the simulated tissue as an interconnected mesh of discrete cells, as it exists *in vivo*. Such approaches are capable of modelling the intermittent behaviour caused by variances in cell coupling, which is especially significant in diseased tissue. Indeed, such discrete, or *network*, models are described by Henriquez and Papazoglou [1996], Rudy [2000], van Capelle and Durrer [1980]. Figures 1.4, 1.5 and 1.6 show crude examples of discrete, cellular automata, models. The majority of models, however, view the tissue as a syncytium, composed of one or two continuous domains. Such a view requires that cell size is sufficiently small with regard to observed phenomena for the tissue to be viewed, and that the units of discretisation are sufficiently large relative to individual cells to avoid modelling the discrete nature of the tissue [Sundnes et al., 2006].

1.7.1 Bidomain

The bidomain model, originally formulated by Tung [1978], considers the intra- and extracellular spaces as discrete, interpenetrating domains, divided by an excitable membrane. The transmembrane potential, V_m, is expressed as the difference between the potentials of each domain:

$$V_m = \phi_i - \phi_e \tag{1.6}$$

The bidomain model is expressed as a system of coupled parabolic (1.8) and elliptic (1.7) equations:

$$\nabla \cdot (\mathbf{G}_i + \mathbf{G}_e)\nabla\phi_e = -\nabla \cdot (\mathbf{G}_i \nabla V_m) \tag{1.7}$$

$$\nabla \cdot (\mathbf{G}_i \nabla V_m) + \nabla \cdot (\mathbf{G}_i \nabla \phi_e) = -S_v I_m \tag{1.8}$$

$$I_m = C_m \frac{\mathrm{d}V_m}{\mathrm{d}t} + (I_{ion} + I_{applied}) \tag{1.9}$$

where subscripts i and e indicate intra- and extra-cellular quantities respectively; ∇ is the gradient operator; $\mathbf{G}$ the conductivity tensors; S_v is the surface-to-volume ratio of a cell; and I_{ion} is the current flow through the membrane per unit area. The kinetics of I_{ion} will be described by a cell model, discussed above, and take the form of a system of ODEs to be solved at each point in the tissue. $I_{applied}$ describes the extent of external stimuli. The bidomain equations conserve charge by ensuring that what flows out of one domain necessarily flows into the other.

1.7.2 Monodomain

By assuming that both domains are equally anisotropic (i.e. $\mathbf{G}_i = k\mathbf{G}_e$ where k is a constant scalar), or by ignoring the extracellular electric field, (System 1.7–1.9) can be reduced to give the monodomain model

$$\frac{\partial V_m}{\partial t} = \nabla \cdot \mathbf{D}\nabla V_m - \frac{I_{ion} + I_{applied}}{C_m} \tag{1.10}$$

where $\mathbf{D} = \frac{Gi}{S_v C_m}$ is the diffusion tensor that describes the conductivity of the medium. The signs of the currents show an *extracellular* perspective, i.e. that the flow of a positive ion from the intra- to the extra-cellular space is positive.

1.7.3 Boundary Conditions

Both bidomain and monodomain tissue models are computed together with boundary conditions, which are usually no-flux Neumann conditions of the form

$$\left.\frac{\partial \mathbf{u}}{\partial \mathbf{n}}\right|_{\Gamma} = 0 \tag{1.11}$$

where $\mathbf{u}$ is the vector of dynamic variables subject to diffusion and $\mathbf{n}$ is the vector normal to the domain boundary, Γ.

1.7.4 Anatomically Realistic Geometry of the Heart

Modelling the geometry of the heart requires knowledge of both its shape, to implement boundary conditions, and its underlying tissue structure, to implement anisotropic diffusion. In particular, it is necessary to capture the orientations of fibres and sheet planes in order to inform anisotropic diffusion. As discussed in Section 1.2.2, the structure of cardiac tissue makes the conductivity of

the myocardium strongly anisotropic. The end-to-end connection of myocytes makes conduction far greater along the muscle fibre. Conduction between adjacent fibres is reduced, and conduction between adjacent sheets is less still. Several models of cardiac tissue geometry have been developed by *ex vivo* studies of histology [Vetter and McCulloch, 1998, Nielsen et al., 1991], MRI [Scollan et al., 2000, 1998] and, more recently *in vivo* DTMRI [Sermesant et al., 2006]. A comparison of DT-MRI and histological methods is given by Ayache et al. [2009].

1.8 Reconstructing ECG

Multi-scale studies, such as those described by Gulrajani [1998], Clayton and Holden [2002], Sundnes et al. [2006] have sought to estimate electrocardiogram (ECG) measurements by placing a fine-grained model of the heart into a coarse-grained model of the torso. Studies of this kind offer a link between the sub-cellular detail provided by *in silico* modelling, and the primary clinical tool for study of the heart.

1.9 Computational Methods

To be solved on a computer, the continuum tissue equations (1.7–1.9), or (1.10) for monodomain, and boundary conditions (1.11) must be discretised in both space and time. The sense in which the word 'discrete' is used here is distinct from the discrete tissue models and discrete states discussed above. For partial differential equations, discretisation is required only to provide a numerical solution. Two methods are commonly used for space discretisation: finite differences and finite elements.

1.9.1 Space Discretisation

Finite Elements

Finite Element Methods (FEM), well developed in computational fluid dynamics, has been in the mainstream of cardiac modelling in the last 3 decades [e.g. Buist et al., 2003, Franzone and Pavarino, 2004, Franzone et al., 2006, Hooks et al., 2002, Sundnes et al., 2006, LeGrice, 2001, Rogers and McCulloch, 1994]. Using FEM, the solution domain is divided into a number of polygonal — commonly triangular (2D) or tetrahedral (3D) — elements. One of the major attractions of FEM is its treatment of irregular domains such as the heart. Because the size of the elements may differ, it is possible for meshes to have more, smaller elements near regions of interest, such as rapidly changing surfaces.

The variability of finite element meshes requires considerable care in the discretisation of any given geometry. A broad selection of software tools now exist to generate finite element meshes.

In finite elements, the solution to the continuous equations is approximated by a set of piecewise polynomial basis functions on the whole domain, thus implicitly incorporating the discretisation of the boundary conditions (1.11).

The result of this approximation will be a system of linear equations for the values of the basis functions at the points, or *nodes* of the discretised mesh, which has to be subsequently solved by linear algebra tools.

The solution matrix of the discretised linear system is typically sparse and asymmetric. It is difficult to construct iterative methods for general classes of non-symmetric problems.

Finite Differences

In Finite Differences, the entire domain is divided into cells, or voxels by a structured, Cartesian grid. Continuum equations are written in terms of finite difference equations. At each interior point, derivatives are approximated by difference quotients over a small interval, i.e. $\frac{\partial \mathbf{u}}{\partial x}$ is replaced by $\frac{\Delta \mathbf{u}}{\Delta x}$ where Δx is small. For example,

$$\frac{\partial^2 \mathbf{u}}{\partial x^2} \approx \frac{(\mathbf{u}_- - 2\mathbf{u}_\cdot + \mathbf{u}_+)}{\Delta x^2} \tag{1.12}$$

where $\mathbf{u}_\cdot$ is the central point, $\mathbf{u}_-$ and $\mathbf{u}_+$ are x-axis neighbours in the negative and positive directions, respectively, and Δx is the space step.

The solution matrix of the discretised system is a trilinear, banded matrix. The banded matrix relates to the limited neighbourhood of points in the FD mesh. Matrices of this kind are inherently simpler to solve than the sparse asymmetric matrices created by FEM. Regular meshes also lend themselves to implicit methods such as the Crank-Nickelson method. Another attraction of Finite Differences is the relative simplicity of coding finite difference equations.

When simulating the anatomically realistic geometry of the heart, integration of data from MRI is more easily used in Finite Differences schemes. Clayton and Panfilov [2008] state that "[s]ince MRI images are typically obtained in tissue slices and assembled into a 3D volume representation, this approach lends itself to a description based on a Cartesian grid."

Neumann boundary conditions in finite differences can be maintained by the introduction of 'fictitious' points at points immediately outside the domain. By assigning these points values from their immediate interior neighbour. In irregular domains, where $\mathbf{n}$ does not lie parallel to a coordinate axis, several methods exist for computing boundary conditions. Some methods, such as that described by Noye and Arnold [1990] and Dumett and Keener [2003] require additional information about the precise location of the boundary closest to the irregular point, which are problematic for cardiac meshes, where the discretisation ordinarily matches the resolution of the measurement equipment used. Buist et al. [2003] describe the use of a three point one-sided difference scheme to be used in the case where one point is missing in the direction of interest. The 'phase-field' method introduced by Fenton et al. [2005] approximates additional resolution at the boundary, by introducing weighted points in a manner similar to image anti-aliasing.

1.9.2 Time Discretisation

As discussed by Keener and Bogar [1998], there are four common timestep methods:

1. Fully explicit (forward Euler):

$$\mathbf{u}_{t+1} = \mathbf{u}_t + \Delta t \left[\mathbf{D}\mathbf{u}_t + f(\mathbf{u}_t)\right]. \tag{1.13}$$

2. Fully implicit (backward Euler):

$$\mathbf{u}_{t+1} - \Delta t \left[\mathbf{D}\mathbf{u}_{t+1} + f(\mathbf{u}_{t+1})\right] = \mathbf{u}_t. \tag{1.14}$$

3. Mixed explicit-implicit (semi-implicit):

$$\mathbf{u}_{t+1} - \Delta t \mathbf{D}\mathbf{u}_{t+1} = \mathbf{u}_t + \Delta t f(\mathbf{u}_t). \tag{1.15}$$

4. Crank and Nicolson [1996]:

$$\mathbf{u}_{t+1} - \tfrac{1}{2}\Delta t \mathbf{D}\mathbf{u}_{t+1} = \mathbf{u}_t + \Delta t \left[\tfrac{1}{2}\mathbf{D}\mathbf{u}_t f(\mathbf{u}_t)\right]. \tag{1.16}$$

The fully explicit method is broadly understood to be easy to implement. It also lends itself well to parallelism. To ensure numerical stability in Finite Differences, step size is critical. Numerical stability requirements of the fully implicit method insist that, for finite differences, $\mathbf{D}\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2d}$, where d is the dimensionality of the medium. Highly non-linear systems may require that $\mathbf{D}\frac{\Delta t}{\Delta x^2} \leq\leq \frac{1}{2d}$. The numerical stability of a discretisation can be tested by running a simple simulation with varying step sizes and examining the results for numerical artefacts. For Finite Elements, a similar numerical stability condition relates the timestep to element size. Consequently, explicit schemes are expensive to compute in fine-grained meshes. While the fully explicit method computes the future state of the system from its current state, the fully implicit method computes the future state of the system based on both the current and future states. The fully implicit method is not restricted by timestep, but does require additional computation, and effort to implement, due to the large non-linear system of equations to be solved at each step. Semi-implicit methods offer improved numerical stability over the fully explicit method, allowing longer timesteps to be used, but require only the solution of a linear system of equations at each step.

1.10 Software Implementation

Computer simulations of cardiac tissue involve vast scale ranges, with activity at μm and μs scales being observed in several cm^3 of tissue over tens of seconds. Moreover, modern biophysically realistic models have brought about an order of magnitude increase in both space and time complexity. The complexity is further increased as simulations expand to include far greater detail of heterogeneous tissue structure and cell types, on increasingly large meshes. In this context, it is unsurprising that the timely completion of simulations relies on modern High Performance Computing (HPC) hardware.

Use of HPC facilities, although essential, is limited by the ad-hoc manner of software development seen across the research community. This can be compounded in situations where funding is more easily obtained for hardware purchase than for the accompanying skilled staff. The result is that many small software development projects are undertaken in isolation, each supporting a particular short-to-medium-term aim. Code developed in this environment is often poorly structured and documented — hampering reuse within the lab — and is unlikely to be portable, preventing use on different platforms elsewhere. Results produced from such software are difficult to compare or repeat, limiting the scope of peer review. This rather insular approach to development excludes a great many members of the community for whom code-level involvement with simulations is neither desirable nor practical. The disparity of cardiac simulation software at present is, arguably, impeding scientific progress.

1.10.1 Shared Memory Parallelism

Shared memory parallelism (SMP) involves multiple threads of execution, each of which have access to a single, shared memory resource. Each thread may read and write the same regions of memory, unless explicitly prohibited from doing so. This approach avoids the need for explicit communication between processes, but requires that developers take great care to avoid race conditions. By default, shared memory programs run in a single, *master*, thread, until a parallel region is entered, at which point additional threads of execution are spawned. The developer must explicitly define parallel regions in which work may be shared between threads. The execution of a multithreaded program is illustrated in Figure 1.8.

Several frameworks exist to support multithreaded programming, such as OpenMP and

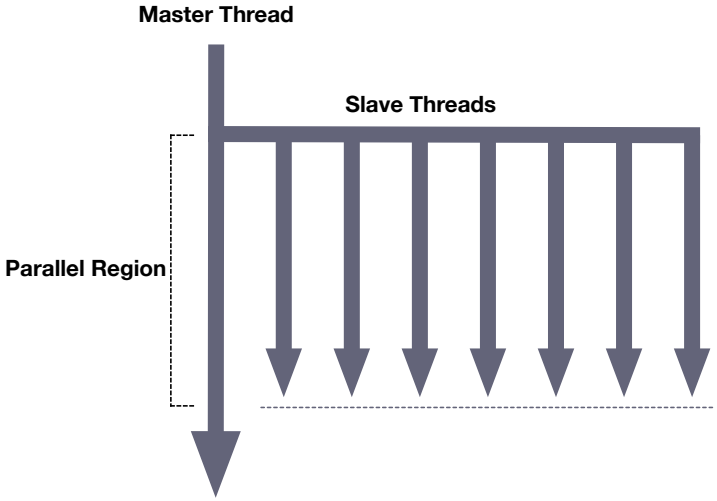


Figure 1.8: Execution of a multithreaded program. A master thread persists throughout execution, while new threads are created as needed.

POSIX Threads. Also, many modern programming languages such as Java and Python implicitly support threads. For HPC applications, the OpenMP library for C/C++ and FORTRAN is popular.

By definition, shared memory applications require that all threads have access to a local block of main memory. In hardware terms, this requires that threads are assigned to multiple processors or processor cores of a single machine. Despite the advent of multicore processors and the increasing number of cores per processor, there exist physical limits with regard to the number of processor cores that can practically surround a single block of main memory and still maintain reasonable access times. This limits the number of threads that may be run concurrently, and ultimately the scalability of parallel codes run on such machines.

For very data-intensive applications, shared memory machines may also lack sufficient RAM to hold all of the necessary program data in memory at once.

1.10.2 Distributed Memory Parallelism

In Distributed Memory Parallelism (DMP), the program is executed on multiple *processes*. A process is a thread of execution with its own allocation of memory and no direct access to that of other processes. Where processes need to exchange data, communication is required via explicit send and receive actions. Unless otherwise specified, all program code runs on every process. This requires DMP processes to be restricted to operating on specific tasks or sets of data to prevent redundant processing. The execution of a parallel program is illustrated in Figure 1.10.

In DMP, an executable runs in one or more processes, where each process has its own, ring-fenced area of memory in which to operate. In a distributed memory machine, several conventional servers, referred to as nodes, are connected to a network. Traditionally, a process maps to a single node. With multi-processor and multicore machines, however, it is likely that each processor or processor core (hereafter referred to as cores) will handle a single process. Figure 1.11 illustrates the hardware architecture of a distributed memory machine with multicore processors. Utilising all of the available cores in this way is likely to yield improved performance,

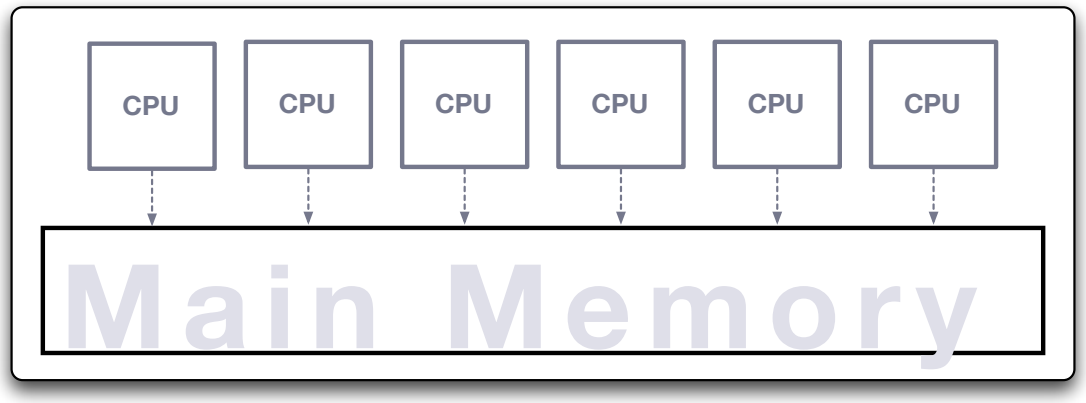


Figure 1.9: Schematic of a typical Shared Memory machine's architecture. Each CPU is has equal access to a single, shared block of main memory.

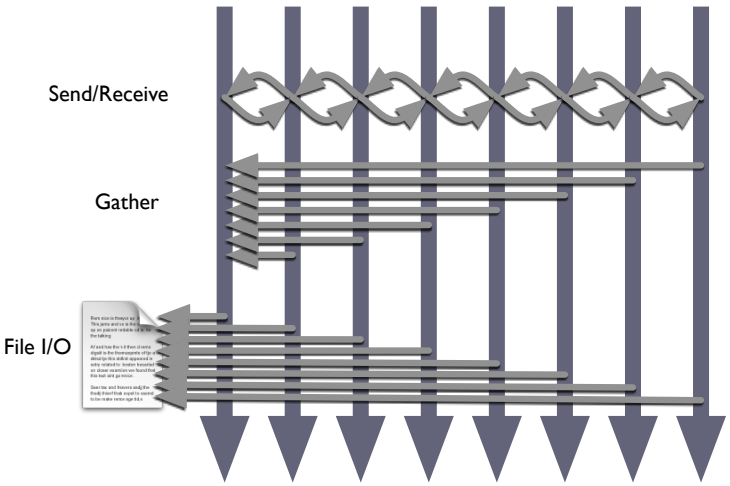


Figure 1.10: Execution of a DMP program. All processes run throughout execution. Work is divided between processes to create speedup. Where coordination is required, processes communicate explicitly. All processes have access to shared I/O resources.

as intra-node communication between processes is likely to be significantly faster than communication across the network. In very memory-intensive applications, where copies of large data structures must be held by every process, the available memory per-core may prevent every core in the node from being used. To combat this, a hybrid of intra-node SMP and intra-node DMP can be considered [Smith, 2000].

While several distributed memory platforms have been developed, MPI [Message Passing Interface Forum, 2003] has become the *de facto* standard for HPC.

1.11 State of the Art Software Packages

1.11.1 Carp

Carp, The Cardiac Arrhythmias Research Package [Vigmond et al., 2003], is a finite-elements based simulation tool, which provides a bidomain tissue model. Carp is available with an open-source license.

Users of Carp configure simulations by 'plugging together' software modules including parabolic and elliptic solvers, preconditioning modules and models of ionic currents. Input is in the form of a set of files specifying the nodes of the Finite Element mesh, the tetrahedra that define it, and the orientation of tissue fibres. In an optional input file, users specify add Purkinje fibres as a tree structure on nodes of the mesh.

The workflow of Carp simulations, including visualisation of simulations, is handled by a dataflow programming environment, allowing the user to graphically connect data processing modules that would otherwise be carried out using separate pieces of software.

Carp has been parallelised for distributed memory machines using MPI.

1.11.2 Continuity

Continuity [McCulloch, 2010] is a simulation tool with particular focus on biomechanics, transport and electrophysiology. Continuity has a client-server architecture, providing a GUI interface on desktop computers, while simulations can be run on the same machine, on a separate server, or in parallel on Linux clusters using MPI.

Continuity is limited to a monodomain tissue model, discretised using finite elements. In addition to the meshes provided with the software, Continuity provides tools to fit finite element meshes to provided datasets, such as those generated from MRI. Continuity is also able to solve biomechanics problems that simulate the deformation of cardiac tissue during contraction.

1.11.3 CHASTE

Chaste, 'Cancer, heart and soft-tissue environment' [University of Oxford Computing Laboratory, 2010] is a generic simulation framework for biological applications, in particular modelling cancer and cardiac electrophysiology. Chaste is made available under an open-source LGPL license.

For cardiac simulations, chaste uses a bidomain tissue model, discretised onto a finite elements mesh. Cell models can be imported using the standardised CellML language. Chaste has also been parallelised for MPI, and provides parallel Input/Output facilities using HDF5. The parameters of Chaste simulations are described using a structured, XML-based language.

Pitt-Francis et al. [2009] make much of the methodologies applied to its development. Using *agile methods* derived from *eXtreme Programming* [Beck and Andres, 2004] and Test-Driven Development [Beck, 2002], their aim is to produce high-quality, alienable software, where 'high

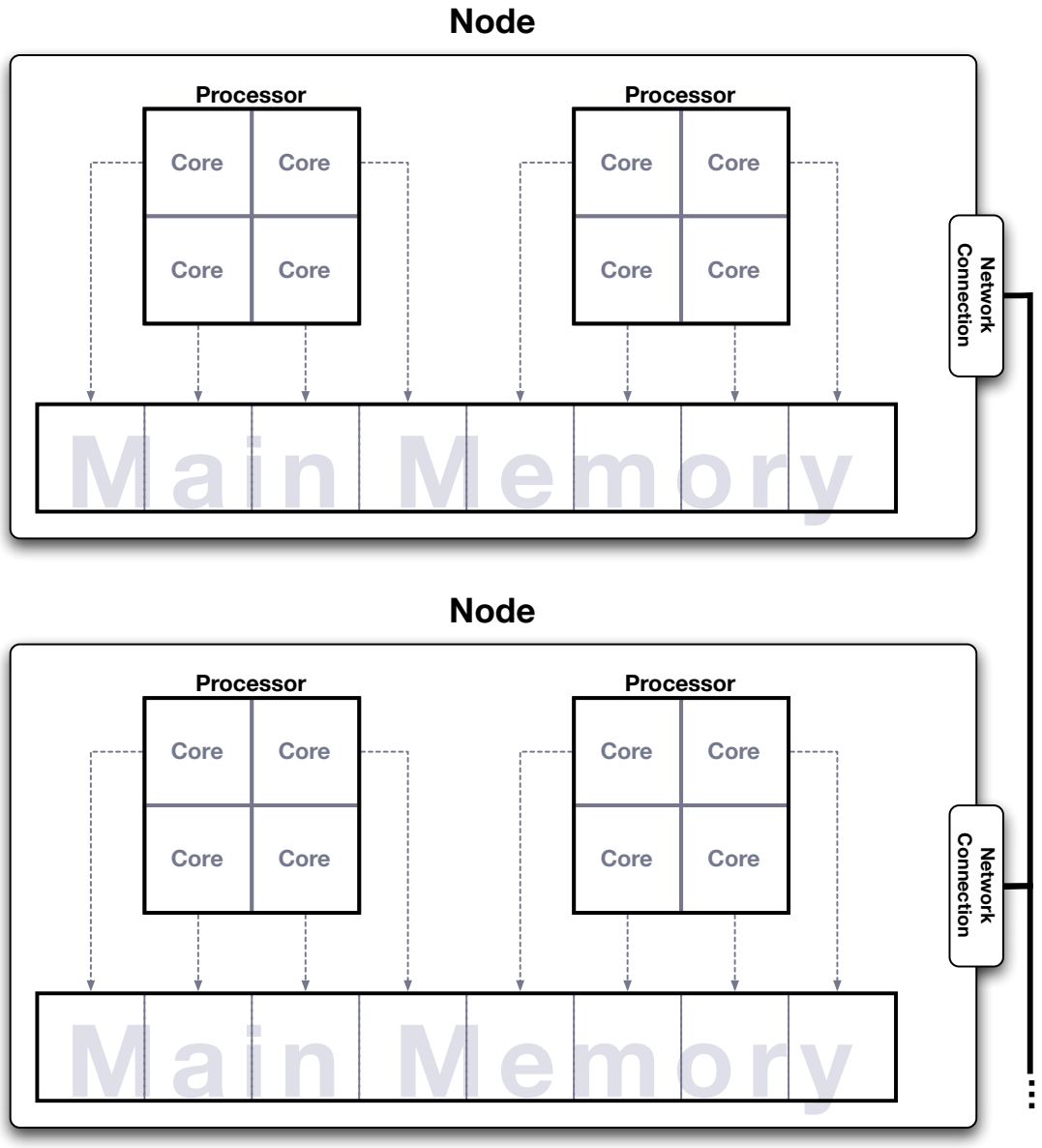


Figure 1.11: Schematic of a typical Distributed Memory machine's architecture. Here, each network-connected node contains two, quad-core processors. Each core runs a single process, which is assigned a region of main memory. No process may address the memory of another, so communication (even intra- node communication) is made by explicitly.

quality' is defined as "extensible, robust, fast, accurate, maintainable and uses state-of-the-art numerical techniques". However, the complexities of Chaste, in particular its XML scripting interface and low-level APIs for extension, do present some difficulty to users.

1.11.4 QUI

The QUI software package [Biktashev and Karpov, Unpublished], developed at the Russian Academy of Sciences, is a modular simulation framework. QUI provides a monodomain tissue model, on cuboid finite-differences meshes.

QUI's user interface is in the form of a bespoke scripting language in which the user lists the software modules, called 'devices', to be used and their corresponding parameters. The script is read and interpreted at runtime to build the simulation. This breaks the edit-recompile-debug-repeat cycle found in much ad-hoc software and prevents the confusion and danger associated with exposing users to the program's source code.

QUI's main solver has been parallelised for shared memory architectures using OpenMP [Wolfe, Unpublished].

Although QUI has been freely available for the past two decades, it has never been documented, either for users or third-party developers.

1.12 The Beatbox Project

This thesis introduces Beatbox, a new computer simulation environment for cardiac problems. Beatbox expands on the QUI package to add distributed memory parallelism and anatomically realistic tissue geometry. This thesis contributes to knowledge on the design and development of High Performance Computing software for the study of cardiac electrophysiology, introducing new techniques and outlining aspects of best practice for performance and usability.

1.12.1 The Case for Extending QUI

The QUI software package was chosen as a basis for Beatbox as it is a rare instance of a robustly developed and alienable simulation code. QUI's use of finite differences should both simplify development and improve the scope for parallelism and performance. QUI is written in the C programming language, which is commonly used for HPC software.

Another attraction of QUI is its in-built script interpreter, which allows users to specify simulations using a domain-specific language. The script specifies the sequence of computational tasks to be run at each time-step of the simulation, such as computing diffusion or producing snapshot images of the medium. Aside from some core functionality, all tasks in the simulation are performed by modular components called *devices*, each of which specifies its required parameters to be parsed from the script at runtime. In addition to constants, parameter values may be variables, arithmetic expressions, or user-defined functions. Such flexibility allows simple comparison of different models, parameter settings and output methods. By allowing simulations to be composed in this way, QUI's modularity reflects that of cardiac models in general. Runtime-interpreted scripts also offer simple exchange of simulation configurations between Beatbox users. Collaborators may quickly and easily adapt and transfer scripts, knowing that their recipient will be able to easily understand and run them. In addition, Beatbox presents an opportunity to accelerate and deepen the peer review process, by opening simulations to scrutiny.

Starting from QUI, Beatbox aims to provide additional functionality, including efficient use of HPC hardware, while retaining QUI's modularity and ease of use. Wherever possible, users

should be shielded from the details of Beatbox's implementation, and able to describe their simulations in domain-specific terms.

1.13 Thesis Outline

The lack of documentation for QUI means that, despite its relatively user-friendly interface, it is unapproachable for many potential users. The first task of the Beatbox project was to study and document the QUI code, describing its scripting interface and APIs for extension in detail. **Chapter Two** provides the first complete documentation of the QUI software package, describing its purpose, design and implementation in detail. Beginning with a study of QUI's scripting language, we examine the structure of simulations, and the modular software components, 'devices', that comprise them. The architecture of QUI's devices are discussed, to both elucidate their operation and establish a template for extension. QUI's runtime operation is discussed through the study of several of its most commonly used devices. Finally, we establish a list of requirements for third-party extensions to QUI.

Chapter Three discusses the development of Beatbox, a parallelised extension of QUI, using MPI. The chapter covers the details of domain decomposition, and describes the way in which QUI's devices have been adapted to benefit from parallelism. Devices that required explicit parallelisation are discussed in detail. In particular, discussion of the `k_func` device forms the basis of an understanding of the types of user-defined function that may be safely executed in parallel. Tests of Beatbox's parallel performance are described and discussed. Finally, we discuss the requirements for third-party extensions to Beatbox, bearing parallel-safety in mind.

Chapter Four describes the addition of a new feature to Beatbox — anatomically realistic geometry. The chapter begins with a discussion of the mathematics of irregular tissue geometry, including anisotropic diffusion. We describe the way in which anatomically realistic meshes can be described using 'geometry files' and their consequent representation in software. We then discuss the implementation of boundary conditions and diffusion for anatomically realistic meshes, comparing possible techniques. The parallel performance of Beatbox using anatomically realistic meshes, with and without anisotropy, is compared and discussed. Finally, we revise the requirements for third-party extensions to Beatbox when using anatomically realistic meshes.

CHAPTER

TWO

THE QUI SOFTWARE PACKAGE

2.1 Overview

The QUI[1] software package [Biktashev and Karpov, Unpublished] was originally developed at the Autowave Laboratory at Puschino, part of the Russian Academy of Sciences, with the purpose of providing an adaptable computing platform with which to investigate the properties of Reaction-Diffusion systems. Since its original development, QUI has been used and adapted to simulate cardiac arrhythmias. QUI uses the finite difference method to simulate the electrical activity in cardiac tissue. QUI allows the user to construct simulations from their component parts by providing cell models, computational functions and output routines that may be combined flexibly.

The fundamental paradigm of QUI is that the simulation process is represented as a loop of 'devices' which are individual modules that perform specific computational, input/output or control tasks. Each type of device can be used more than once in the same loop, thus providing more than one device. This loop of devices is constructed at runtime, based on instructions given in the input script. The script describes the devices used in a particular simulation and their parameters, in a domain-specific scripting language of flexible syntax including, e.g. arithmetic expressions, recursive calls of other scripts, etc. QUI was only designed and used for monodomain models in simple Cartesian coordinate lattice.

2.1.1 Modular Structure

As discussed in Section 1.5, cardiac simulations are usually comprised of a simulation medium (a mesh with dynamic variables at each point), initial and boundary conditions, and a sequence of operations to be performed on the simulation medium. This sequence commonly involves a solver for the tissue model, computation of the Laplacian, computations to maintain boundary conditions and some manner of output. In QUI, each of these tasks is performed by a software module known as a device. A QUI simulation consists of a set of devices with access to the simulation medium. Conceptually, the devices for a simulation are arranged as a 'ring', as illustrated by Figure 2.1. Running the simulation iterates over the ring of devices, calling each device in turn to perform its part of the task. One 'revolution' of the ring represents one step forward in simulation time.

[1] Although QUI is an acronym, there is no record of what it stands for.

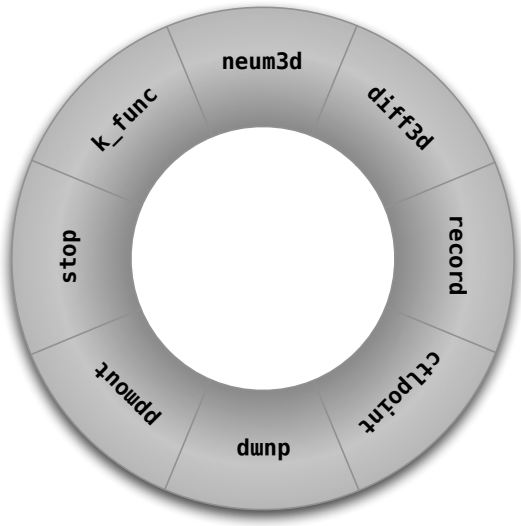


Figure 2.1: Conceptual view of the Ring of Devices. The ring is comprised of devices requested by the user. At runtime, each device is executed in turn. Each 'revolution' of the ring represents one timestep.

QUI provides a set of in-built devices to perform common simulation tasks. Devices may be parameterised by the user. All devices accept a set of common parameters, such as when they should run, or the region of the mesh on which they should operate, and most devices accept additional parameters to offer the user greater control and flexibility.

Advanced users can extend QUI's functionality by implementing new devices, and QUI provides a well-structured interface to do this.

2.1.2 Running Simulations with QUI

User input to QUI is in the form of a script file, which describes the dimensions of the mesh, sets initial conditions, and specifies the ring of devices along with their parameters. Figure 2.2 illustrates the main activities that take place when QUI is run. The process is described below.

1. The user script is read and the simulation initialised:
   - The simulation medium is created with the specified dimensions.
   - Devices are added to the ring of devices with the specified parameters.
2. The simulation is run:
   - Each device in the ring is called in turn to perform its part of the simulation task.
   - Repeat until the stop condition is met, or an error occurs.
3. The simulation is terminated.

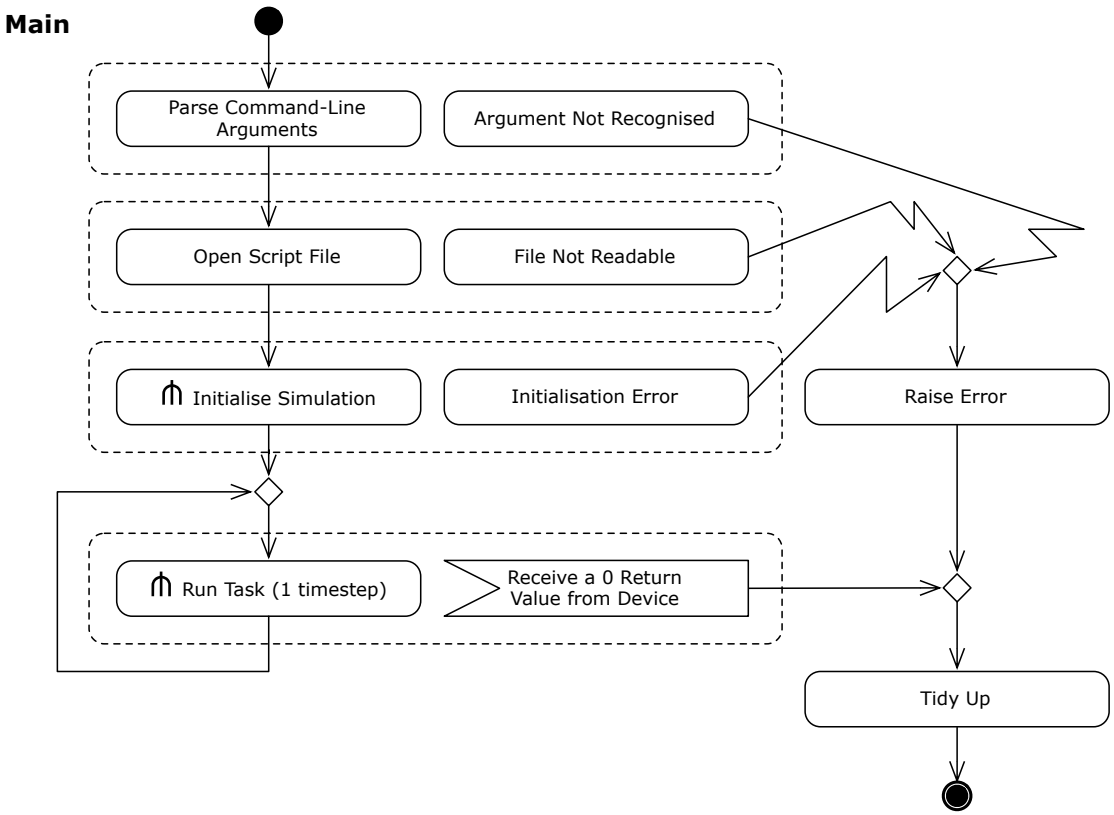


Figure 2.2: Top-level UML activity diagram showing QUI's flow of execution.

2.1.3 Chapter Structure

This chapter aims to describe how QUI can be used to run simulations of cardiac activity, and to elucidate the details of its main operations. Section 2.2 begins with an explanation of QUI's user interface, its scripting language. Section 2.3 describes the software construction of QUI simulations by introducing its principal components, devices, and their related datatypes. Sections 2.4–2.6 describe the runtime functionality of QUI, examining how the script is interpreted to build simulations, and how those simulations are run and ultimately terminated. Sections 2.7–2.12 introduce some of the more commonly used QUI devices, including how they can be controlled from the script, and some details of their operation. Finally, Section 2.13 discusses the ways in which third-party developers can augment the functionality of QUI by developing custom devices, or adding new models of cardiac myocytes' kinetics.

The explanation of QUI given in this chapter is not an exhaustive one; it gives sufficient detail on which to base one's understanding of the parallelisation discussed in Chapter 3. One particular omission is in the area of the on-screen display. QUI provides functionality for some aspects of a simulation to be visualised on screen using X11 and OpenGL libraries. The visualisation of cardiac simulations, while vitally important to the field, is beyond the scope of this thesis and, as such, no attempt is made to study, incorporate, or extend QUI's visualisation features.

2.2 QUI Scripts

A QUI simulation is defined by the user in the form of a script. A script is a text file, written in a proprietary language, which specifies the parameters of the simulation. By using scripts, QUI prevents users from having to change the program's source code in order to parameterise simulations and breaks the edit-debug-build cycle common to the use of ad-hoc simulation tools. Scripts also provide a simple means of exchanging simulation data between collaborators and, if desired, for peer review. Scripts offer a human-readable view of a simulation, which are decoupled from QUI's implementation such that any script can be run on any of QUI's supported hardware platforms. This section introduces QUI scripts by first examining the scripting language, before discussing common applications.

Unlike interpreted 'scripting' languages such as PHP [The PHP Group] or Python [Python Software Foundation], QUI scripts are not run during the simulation. The script is read once to build the simulation, after which script code does not define the flow of execution. Section 2.11 discusses how the user can define functions to be executed at runtime using the `k_func` device.

2.2.1 The QUI Scripting Language

Similar to other programming languages, QUI scripts allow the user to define variables and macros, include external code and make calls to the operating system. The QUI scripting language also provides a small library of arithmetic and logical functions. The user may also improve the legibility of their code, or disable portions of the code using comments. Each of these features is discussed below.

A QUI script comprises a series of *commands*. A command takes the form of any number of lines of script code beginning with a recognised keyword and ending with a semicolon (`;`). The following code excerpt shows examples of three commands:

```
def int sideLength 26;

state xmax=sideLength, ymax=sideLength, zmax=sideLength, vmax=3;
```

```
euler
    v1=[iext]
    ode=lrd
    par={ht=ht IV=@24}
;
```

The types of command in a QUI script are listed below. Each command describes an action to be taken by QUI, with the keyword at the beginning of the command being the verb.

- `rem` — A 'remark'. Marks code as a comment. These are ignored by the interpreter.
- `if` — Allows a script command to be conditionally read. This does not affect the flow of execution as the simulation is run, but operates much like conditional compilation.
- `def` — Defines a variable or macro.
- `state` — Sets the dimensions of the simulation medium.
- `screen` — Sets parameter for on-screen display.
- Device calls — Adds a device to the ring of devices. The keyword in a device call may be any of the device names listed in *devlist.h*, e.g. `euler`.
- `end` — Marks the end of the script.

2.2.2 Preprocessing

As each line of the script is parsed, QUI preprocesses the code in a manner similar to the C Preprocessor [see Kernighan and Ritchie, 1988, chap. 4]. The QUI preprocessor handles four tasks: skipping comments, including other QUI scripts, calling system commands and expanding string macros. Each of these are discussed below.

Skipping Comments

Comments can be added to QUI scripts in three ways:

```
/*
    Multi-line
    C style
    comment
*/

// Single-line C++ style comment

rem 'Remark' command-style comment, which must end with a semicolon;
```

Code in a comment is ignored by the parser.

Including Other QUI Scripts

Where the relative path to a QUI script file is enclosed in in angle brackets (`< >`), the content of the referenced file will be read and inserted at that location. Unlike in the C programming language, the name of the file in angle brackets is not preceded by an `#include` command. This can be used to maintain consistency across a number of simulations, or to reduce code redundancy. QUI

replaces a filename in angle brackets with the file's entire contents. For example, given a script called *useful.qui*:

```
// Here is a lot of useful code...
```

Listing 2.1: *useful.qui*

and a script that includes *useful.qui* as follows:

```
// This is my own script
<useful.qui>
// Here's some more of my own code.
```

Listing 2.2: A script that includes *useful.qui*.

The result after preprocessing is shown below:

```
// This is my own script
// Here is a lot of useful code...
// Here's some more of my own code.
```

Listing 2.3: Effective resultant script, after *useful.qui* has been included.

Calling System Commands

QUI replaces code in backticks (‘‘) with the result of that code when run as a system command, via `system()` (*stdlib.h*). For example ‘`date '+%Y%m%d-%H:%M:%S'`‘ will be replaced with the result of the UNIX `date` command.

Expanding String Macros

String macros allow the user to define reusable strings that can be `pasted` throughout a script. String macros are distinct from variables in that they are expanded once, prior to the script being interpreted and cannot therefore be assigned values other than when they are defined. A string macro is defined as follows:

```
def str <macro name> <value>;
```

where `<macro name>` is a string using letters, numbers, underscores (`_`) or hyphens (`-`) and `<value>` is any string not including a semicolon.

When defined, the macro's name, wrapped in brackets (`[ ]`) is associated with its value. When QUI finds a macro's name in square brackets, it replaces them with the value. In the following excerpt, the variable `hat` is assigned the value `porkpie`. The variable `headware` is assigned the value `hat`.

```
def str snack porkpie; // Assigned string 'porkpie'.
def str hat [snack]; // Assigned string 'porkpie'.
def str headware hat; // Assigned string 'hat', not value of hat macro.
```

A macro can expand to anything that could be typed in the script. For example, a string macro can be used in place of an `int`, `long` or `real`:

```
def str ninetynine 99; // Assigned string value '99'.
def int number [ninetynine]; // Assigned integer value 99.
```

A number variable cannot, however, be used to define a macro:

```
def int number 99; // Assigned integer value 99.
def str ninetynine number; // Assigned string value 'number'.
```

It is possible to assign several lines of code (excluding semicolons) to a string macro, so this:

```
def str instruction ppmout
when=out file="ppm/%04d.ppm" mode="w"
r=[u] r0=umin r1=umax
g=[v] g0=vmin g1=vmax
b=[i] b0=0 b1=255;
[instruction];
```

is equivalent to:

```
ppmout
when=out file="ppm/%04d.ppm" mode="w"
r=[u] r0=umin r1=umax
g=[v] g0=vmin g1=vmax
b=[i] b0=0 b1=255;
```

2.2.3 Defining Arithmetic Variables with the `def` Command

A variable is defined using the `def` command, like this:

```
def <type> <variable name> <initial value>;
```

where `<type>` is one of QUI's pre-defined datatypes:

`int` Integer. Equivalent to `int` in C.

`long` Long integer. Equivalent to `long` in C.

`real` Real number. Equivalent to `double` in C.

`<variable name>` is a string using letters, numbers, underscores (`_`) or hyphens (`-`). Variables defined in this way are accessible in the script and in user-defined functions run by the `k_func` device. `<initial value>` (optional) is an expression that may be evaluated to the correct type. Expressions are discussed in greater detail below.

Variables defined in the script are distinct from the C language variables used in QUI's implementation. For clarity, variables defined in the script, using `def` may be referred to as k_variables. QUI stores the names of macros wrapped in their brackets (`[]`), therefore it is possible for a script to define macros and variables with the same name.

Expressions

Any numerical value in a QUI script, be it a variable definition or parameter, may be specified as a mathematical expression. An expression may be a literal value, such as `5.0`, or the result of some computation, such as `6*5` or `count/2`. QUI provides the standard infix mathematical operators: assignment (`=`), addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`). For slightly more complex operations QUI provides a collection of functions, the parameter and return types for which are all `real`:

`atan2(a, b)` Returns the arctangent of `b/a`, using the signs of the arguments to compute the quadrant of the return value.

`hypot(a, b)` Returns the square root of the sum of the squares of `a` and `b`.

`tanh(a)` Returns the hyperbolic tangent of `a`.

`erf(a)` Returns the Gauss error function of `a`.

`if(test, then, else)` Returns `then` if `test` evaluates to true, `else` otherwise.

`ifne0(test, then, else)` Returns `then` if `test` is not equal to `0`, `else` otherwise.

`ifeq0(test, then, else)` Returns `then` if `test` is equal to `0`, `else` otherwise.

`ifgt0(test, then, else)` Returns `then` if `test` is strictly greater than `0`, `else` otherwise.

`ifge0(test, then, else)` Returns `then` if `test` is greater than or equal to `0`, `else` otherwise.

`iflt0(test, then, else)` Returns `then` if `test` is strictly less than `0`, `else` otherwise.

`ifle0(test, then, else)` Returns `then` if `test` is less than or equal to `0`, `else` otherwise.

`ifsign(test, neg, zero, pos)` Returns `neg` if `test` is strictly less than `0`, `zero` if `test` is equal to `0`, `pos` if `test` is strictly greater than `0`.

`gt(a, b)` Returns `1.0` if `a` is strictly greater than `b`.

`ge(a, b)` Returns `1.0` if `a` is greater than or equal to `b`.

`lt(a, b)` Returns `1.0` if `a` is strictly less than `b`.

`le(a, b)` Returns `1.0` if `a` is less than or equal to `b`.

`eq(a, b)` Returns `1.0` if `a` and `b` are equal.

`ne(a, b)` Returns `1.0` if `a` and `b` are not equal.

`mod(a, b)` Returns the remainder of the integer division of `a` by `b`.

`max(a, b)` Returns the greater of `a` or `b`.

`min(a, b)` Returns the lesser of `a` or `b`.

`crop(test, min, max)` Returns `min` if `test` is strictly less than `min`. Returns `max` if `test` is strictly less than `max`. Returns `test` otherwise.

`rnd(a, b)` Returns a random real number greater than or equal to `a`, strictly less than `b`.

`u(x,y,z,v)` Returns the value of dynamic variable `v` at point `(x,y,z)` in the mesh. If non-integer values are given for `x` `y` or `z`, the value is linearly interpolated from surrounding points. Non-integer values for `v` are rounded down.

`dudx(x,y,z,v)` Returns the difference in `v` values at the location with `x` $\pm$`0.5`.

`dudy(x,y,z,v)` Returns the difference in `v` values at the location with `y` $\pm$`0.5`.

`dudz(x,y,z,v)` Returns the difference in `v` values at the location with `z` $\pm$`0.5`.

2.2.4 Defining the Simulation Medium with the `state` Command

The first functional task of the script is to define the mesh. The mesh is stored as a four-dimensional array, corresponding to a three-dimensional, regular Cartesian mesh with an array of dynamic variables at each point. Dynamic variables hold space-dependent values, local to each point in the mesh. The majority of dynamic variables are commonly employed to hold variables of the cell model. The dimensions of the mesh and number of dynamic variables (i.e. the size of the four-dimensional array) are set using the `state` command. `state` has four compulsory parameters; `xmax`, `ymax`, `zmax` and `vmax`. The mesh defined in the example below has 300 points along the x axis and 400 points along the y axis. Since `zmax` is equal to `1`, there is one point along the z axis, making the medium flat on the x-y plane. `vmax` is the number of dynamic variables stored at every point in the mesh.

```
state xmax=300 ymax=400 zmax=1 vmax=3;
```

Listing 2.4: Defining a two-dimensional mesh with the `state` command.

QUI assumes that the simulation medium will follow a hierarchy of axes; x before y before z, i.e. one-dimensional simulations must use the x axis and two-dimensional simulations must use the x and y axes.

2.2.5 Device Calls

Following the `state` declaration, a script can add to the *ring of devices* by invoking or 'calling' QUI devices. A device call takes the form of the device name, followed by parameters as key-value pairs, separated by spaces. The excerpt below shows the `euler` device being called with some parameters.

```
euler v1=[iext] ht=ht ode=lrd pars={IV=@24} name=Geoff;
```

Each device call will add an instance of the device to the ring of devices. It is possible to instantiate the same device, e.g. `euler`, multiple times and the parameters of each instance will remain independent.

Assigning Device Parameters

Device parameters are assigned following the device name as key-value pairs with the syntax `key=value`. Since spaces separate device parameters, an `str` parameter value that includes spaces, should be enclosed in quotes (`" "`). Expressions assigned to `int`, `long` and `real` parameters will be reduced to literal values when read. An example is shown in Listing 2.5.

Some devices may request parameters formatted as a codestring or a block. A codestring is provided in the same way as a string parameter, with the exception that the string must be a valid QUI script *expression*. The expression should compile to the variable type specified in the device's documentation. Codestrings allow devices to evaluate an expression as the simulation is running, rather than reduce it to a literal value.

A block is a string enclosed in braces (`{ }`). The contents of a block will be passed literally to the device, without intervention from the script interpreter. This allows a block to contain characters such as `;` or `"`, that could not be included in a string parameter. Some devices, notably `k_func`, use a block to accept script statements. `euler` uses a block to pass parameter assignments to its RHS module. Examples of both are shown below. Listing 2.5 shows the assignment of a parameter, `bar` to the `foo` device. Although the value of `hat` may change throughout the simulation, the value of the `bar` parameter will be set only once, when the device is called. In this case, `bar` will be assigned the value `40`.

```
def real hat 10.0;
foo bar = min(hat,60)*4;
```

Listing 2.5: Assigning a device parameter.

```
k_func [nowhere] pgm={
    stim = eq(t,stimTime);
};

euler v0=0 v1=neqn-1 ht=ht ode=crn par={ht=ht, IV=@[iext]};
```

Listing 2.6: Assigning block parameters to the `k_func` and `euler` devices.

Generic Device Parameters

A set of generic parameters, listed below, is applicable to all device types. All of the generic parameters listed below are optional — if no value is given for them, a default will be provided by QUI. Generic parameters and their defaults are described below.

`(str) name` Given name for the device. If the script specifies two instances of a device, giving one of the devices a `name` will disambiguate the devices in any output messages. Also, some devices refer to another device in the simulation using its `name` parameter.

`(real) when` The device's *condition*. The value given for `when` must be a variable of type `real`. A literal value cannot be used for this parameter. The device will be run only on timesteps when `when` evaluates to TRUE. For devices that should run on every timestep, QUI provides the system variable `when=always`, whose value is `1`. `always` is the default value for the `when` parameter. A pointer to the condition variable is assigned to the `c` field of the `Device` structure.

Space Parameters These specify the points of the mesh on which the device will operate:

`(int) x0` Lower x bound. Defaults to `X0`.

`(int) x1` Upper x bound. Defaults to `X1`.

`(int) y0` Lower y bound. Defaults to `Y0`.

`(int) y1` Upper y bound. Defaults to `Y1`.

`(int) z0` Lower z bound. Defaults to `Z0`.

`(int) z1` Upper z bound. Defaults to `Z1`.

`(int) v0` Lower v (dynamic variable) bound. Defaults to `0`.

`(int) v1` Upper v (dynamic variable) bound. Defaults to `v0`.

In general, a device will operate on points from, e.g., `x0` $\leq x \leq$ `x1`. In some cases, a device may use space parameters to specify specific points or dynamic variables, as opposed to ranges. For example, QUI's `diff*` devices use `v0` to indicate the dynamic variable containing transmembrane potential and `v1` to indicate where the Laplacian should be assigned.

For devices that have no effect on the simulation medium, convention is to use the string macro `nowhere`, provided by *std.qui*. `[nowhere]` expands to `x0=0 x1=0 y0=0 y1=0 z0=0 z1=0 v0=0 v1=0`.

Window Parameters These specify the placement and colouring of device's on-screen displays.

`int row0` Lower row bound.

`int row1` Upper row bound.

`int col0` Lower column bound.

`int col1` Upper column bound.

`int color` Colour.

2.3 QUI Devices

Devices are the conceptual building blocks from which simulations are constructed. The implementation of each device is split between generic code, provided by *device.h*, and device-specific implementation in a *.c* file, which, by convention, has the same name as the device. Devices in a simulation are manifested in instances of the `Device` structure. These structures associate the functions and persistent data, such as parameter values, required to perform a particular sub-task. The `Device` structure is defined in *device.h*; its fields are described in Table 2.1.

Type	Name	Description
`Condition`	`c`	A k_variable of type `real` that indicates when the device is to be run.
`Space`	`s`	Describes the scope of the device's effects; gives coordinate bounds within which the device may operate on the medium.
`Window`	`w`	Describes the on-screen display of the device.
`Par`	`*par`	The device's *parameter structure*; a device-specific structure that holds persistent data required by the device including its device-specific parameters from the script.
`Proc`	`*d`	The device's Destroy function; frees the device's resources.
`Proc`	`*p`	The device's Run function; carries out the device's part of the task.
`Name`	`n`	The name of the device. Provides a means of referring to devices in QUI scripts.

Table 2.1: `Device` Structure Definition

2.3.1 Condition

The `c` field, which takes its value from the `where` parameter, identifies the timesteps in which the device should be run. The `Condition` datatype is a pointer to a `real` (k_)variable, and is treated as 'boolean' — FALSE if equal to `0`, TRUE otherwise. In order to have the device run at some timesteps but not at others, the variable will usually be updated based on a user-defined function, `k_func`, in the QUI script.

2.3.2 `Space`

Every device contains a `Space` structure, as described in Table 2.2. The `Space` structure is used to constrain the effects of the device to a specific region of the medium, and/or to specific variables

Type	Name	Description
`int`	`x0`	Minimum x value accessible by the device.
`int`	`x1`	Maximum x value accessible by the device.
`int`	`y0`	Minimum y value accessible by the device.
`int`	`y1`	Maximum y value accessible by the device.
`int`	`z0`	Minimum z value accessible by the device.
`int`	`z1`	Maximum z value accessible by the device.
`int`	`v0`	Minimum v value accessible by the device.
`int`	`v1`	Maximum v value accessible by the device.

Table 2.2: Fields of the `Space` structure.

in the vector of dynamic variables. Any device that iterates over variables in the medium can use the fields of its `Space` structure as bounds for loops. For example, the loop below shows a common use for the `Space` structure, `s`:

```
for(x=s.x0; x<=s.x1; x++){
    for(y=s.y0; y<=s.y1; y++){
        for(z=s.z0; z<=s.z1; z++){
            // Do this everywhere in my Space
        }
    }
}
```

Listing 2.7: An example of device code iterating over its space.

2.3.3 Window

The `Window` structure forms part of QUI's on-screen display functionality and, as such, falls outside the scope of this thesis and will not be discussed here.

2.3.4 Parameter Stucture

The device's state is held in its parameter structure, a `struct` pointed to by `par`. Although the requirements of different devices vary, a common data type, `STR`, is defined in each device to meet its specific data storage needs. `STR` is essentially a 'marker' type — there is no generic implementation, it serves merely to signal the intended use of the structure. Within the device's code, its own `STR` structure is referred to using the pointer `S`.

2.3.5 Behaviour

The device's behaviour is defined by two functions, pointed to by fields of the `Device` structure. The `p` field points to the device's Run function, which calls the device to execute its part of the simulation task. The `d` field points to the device's Destroy function, which calls the device to free up any memory it is using.

2.3.6 The Run Function

The template macros RUN_HEAD and RUN_TAIL provide common structure to device's Run function. The RUN_HEAD macro will expand to define a function called run_<name>, where <name> is the argument to the macro, the name of the device. The definition of the RUN_HEAD macro from *device.h* is shown in Listing 2.8.

```
#define RUN_HEAD(name)   \
    PROC(run_##name) {   \
    STR *S = (STR *) par;
```

Listing 2.8: *device.h* — The RUN_HEAD template macro.

The PROC macro in turn is expanded to give the function signature as shown in Listing 2.9.

```
#define PROC(name) int name(Space s,Window w,Par par)
```

Listing 2.9: *device.h* — The PROC macro.

Hence, the Run function has three parameters, all of which are treated as inputs. s is the device's Space structure. w is the device's Window structure. par is the device's parameter (STR) structure. The RUN_HEAD macro casts the Device structure's par pointer to STR and assigns it to local pointer S. This allows the Device structure's generic par pointer to be used to access the device-specific parameter structure. For this reason, code within the Run function should refer to the device's STR structure using the S pointer. Commonly, the Run function begins by retrieving values from the parameter structure. To aid access to fields of the parameter structure, *device.h* provides shortcut macros to declare local variables and initialise them with values from their namesake fields in the parameter structure.

- CONST assigns the value of the structure variable to a local variable.
- VAR assigns the address of the structure variable to a local pointer.
- ARRAY assigns the address of the array pointed to by the structure variable to a local pointer.

Definitions of the above macros are shown in Listing A.3.

The RUN_TAIL macro closes the function, returning 1 to indicate success, as shown in Listing 2.10.

```
#define RUN_TAIL(name) \
  return 1;    \
}
```

Listing 2.10: *device.h* — The RUN_TAIL template macro.

2.3.7 The Destroy Function

The Destroy function is called at the end of the simulation and allows the device to perform any necessary tasks before QUI exits. The DESTROY_HEAD macro expands to assign the parameter structure to the local pointer S. The DESTROY_TAIL macro expands to free the parameter structure and return 1 to indicate success. Between these two macros, devices should free any dynamically allocated memory and close any open files.

Definitions of the DESTROY_HEAD and DESTROY_TAIL macros are given in Listing 2.11 and 2.12, respectively.

```
#define DESTROY_HEAD(name)\
PROC(destroy_##name) {\
STR *S = (STR *) par;
```

Listing 2.11: *device.h* — The `DESTROY_HEAD` template macro.

```
#define DESTROY_TAIL(name)\
  FREE(par);\
  return 1;\
}
```

Listing 2.12: *device.h* — The `DESTROY_TAIL` template macro.

2.3.8 Template Macros

All devices must implement three functions referred to as Create, Run and Destroy. To assist developers in coding to the device interface, and to provide some common functionality, *device.h* defines three pairs of macros that can be used to 'wrap' the implementation code for each function. This practice bears some similarity to the *Template Method* design pattern described by Gamma et al. [1995].

The macros are:

- `CREATE_HEAD` and `CREATE_TAIL`
- `RUN_HEAD` and `RUN_TAIL`
- `DESTROY_HEAD` and `DESTROY_TAIL`

2.3.9 The Create Function

When a device is to be instantiated, its Create function is called. As shown in Listing A.1, the `CREATE_HEAD` macro expands to allocate memory for the device's parameter structure and assigns its address to local pointer `S`. Any further device-specific initialisation, such as assigning device-specific parameters to the parameter structure, can be defined by the developer between the `CREATE_HEAD` and `CREATE_TAIL` macros.

The `CREATE_TAIL` macro defines code to point the fields of the Device structure to the corresponding functions and structures in the device code. Pointers are assigned to the device's run and destroy functions. These allow *qui.c* to use the device as required.

The `CREATE_TAIL` macro expands to assign the device's Run and Destroy functions, discussed below, to the parameter structure. The parameter structure itself is then assigned to the `par` field of the device's `Device` structure. The `CREATE_TAIL` macro is show in Listing A.2.

2.3.10 Name

The device's `n` field is populated from the `name` parameter in the script. The `name` parameter allows the user to provide the device with a custom name. This can be useful in simulations where multiple instances of a device are being used, to identify a particular instance in simulation output.

The name parameter also provides a means by which devices may reference other devices. Using the `get_dev()` function (*qpp.c*), a device can obtain a pointer to a device with the given name. At present, this rather esoteric feature is only implemented in the `clines` device, which uses a reference to the simulation's `euler` device to plot data from its RHS module.

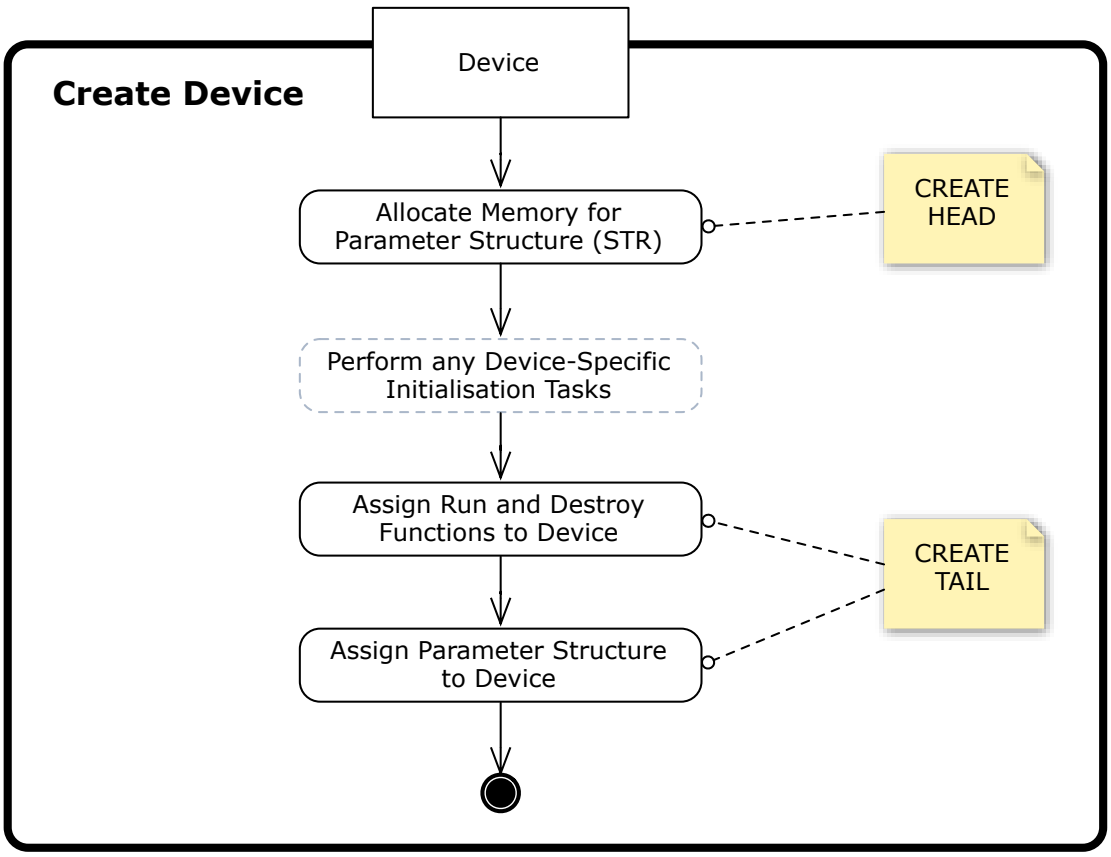


Figure 2.3: High-level UML activity diagram showing device creation. Notes mark the macros defined in *device.h* that define common device behaviour.

2.4 Building the Simulation

2.4.1 The `init()` Function

Initialisation of the simulation is coordinated by the `init()` function (*init.c*). As illustrated in Figure 2.4, `init()` reads each command from the script and performs the corresponding action. This process repeats until an `end` command is read, or the end of the script file is reached. `init` calls the `read_command()` function (*qpp.c*) to read commands from the script. `read_command()` in turn calls `qgetc()`, which performs the preprocessing tasks discussed in Section 2.2.2, before returning the preprocessed command.

Using the `first_word()` function (*qpp.c*), `init()` extracts the keyword at the start of the command and tests the word against its list of acceptable commands, which includes the names of all available devices, included from *devlist.h* and the script keywords; `if`, `rem`, `def`, `state`, `screen`, `end`. The `def` command triggers the definition of a new variable using the `def()` function (*qpp.c*), which is described in Section 2.4.3. The `state` command triggers the initialisation of the `state` module, which builds the mesh, and is discussed in Section 2.4.4. Device calls trigger the creation of the specified device. The generic device creation process is discussed in Section 2.3.9.

2.4.2 QPP — The QUI Preprocessor

The QUI Preprocessor (QPP), defined in *qpp.c* and *qpp.h*, is responsible for parsing and interpreting the QUI script. QPP provides several functions, listed in Table 2.3, for use in extracting data from the QUI script. `init()` uses QPP's `read_command()` function to fetch each device declaration in order to form the ring of devices. When a device is created (see Figure 2.5), the `w` parameter of its Create function is passed a pointer to the portion of the device call that follows the device name, `rest`. `rest` contains any device-specific parameters from the QUI script. From the Create function, `w` can be passed to QPP, together with information about the name and type of value to extract from the string.

Name	Description
`qopen`	Expands a string macro.
`first_word`	Extracts the keyword from a command.
`read_command`	Extracts a command from the script.
`calc`	Evaluates a script expression.
`def`	Defines a k_variable.
`def_dev`	Defines a device's name as a k_variable.
`get_dev`	Obtains a reference to a device from its name.
`accept_condition`	Accepts the device's `when` parameter.
`accept_space`	Accepts the device's space parameters.
`accepti`	Accepts an integer parameter.
`acceptl`	Accepts a long integer parameter.
`accepts`	Accepts a string parameter.
`acceptf`	Accepts a file parameter.
`acceptc`	Accepts a codestring parameter.
`acceptb`	Accepts a block parameter.

Table 2.3: Descriptions of selected QPP functions.

Many devices make use of shortcut macros defined in *qpp.h*, which require fewer, simpler

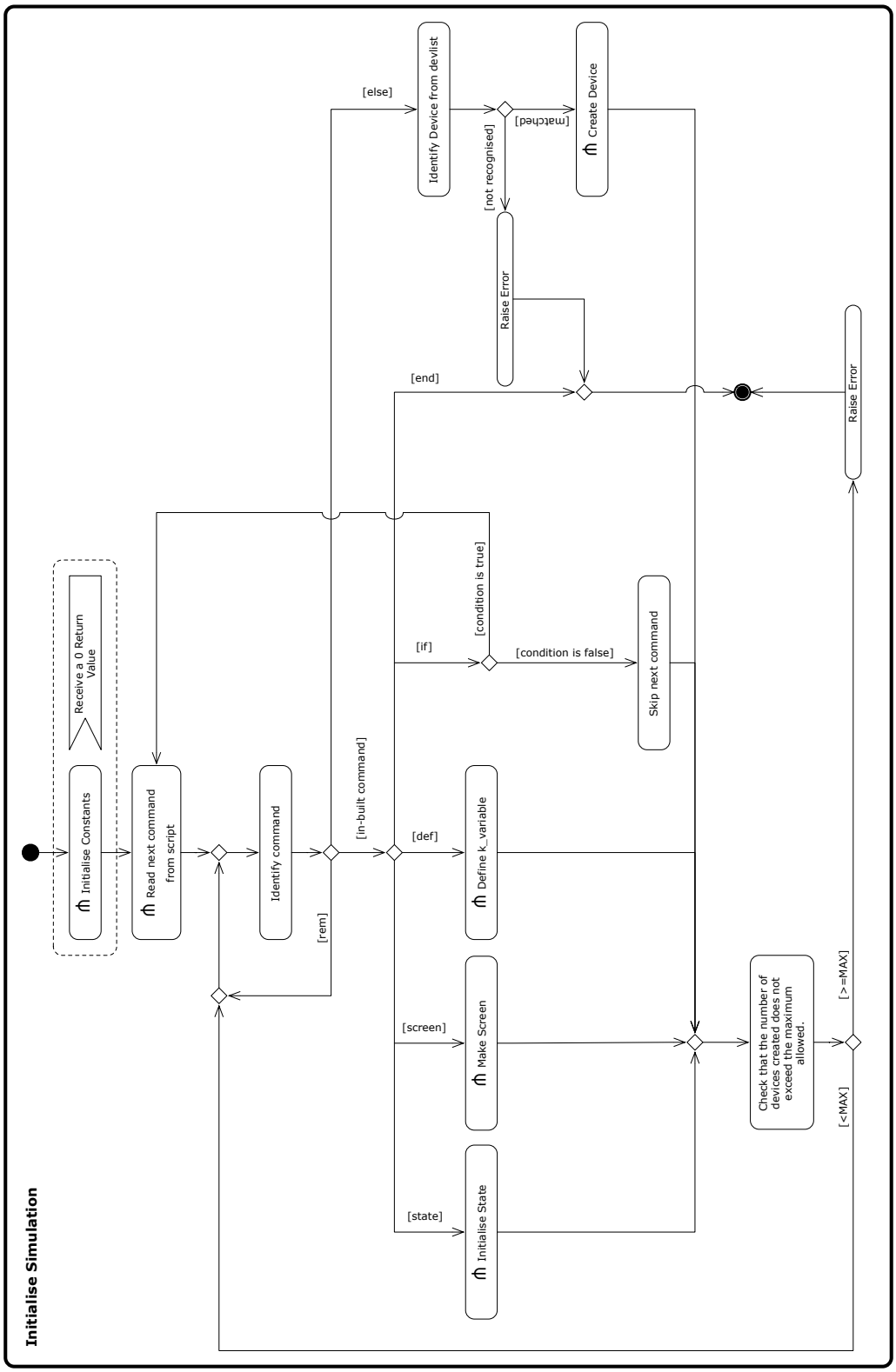


Figure 2.4: UML activity diagram showing the initialisation of the simulation.

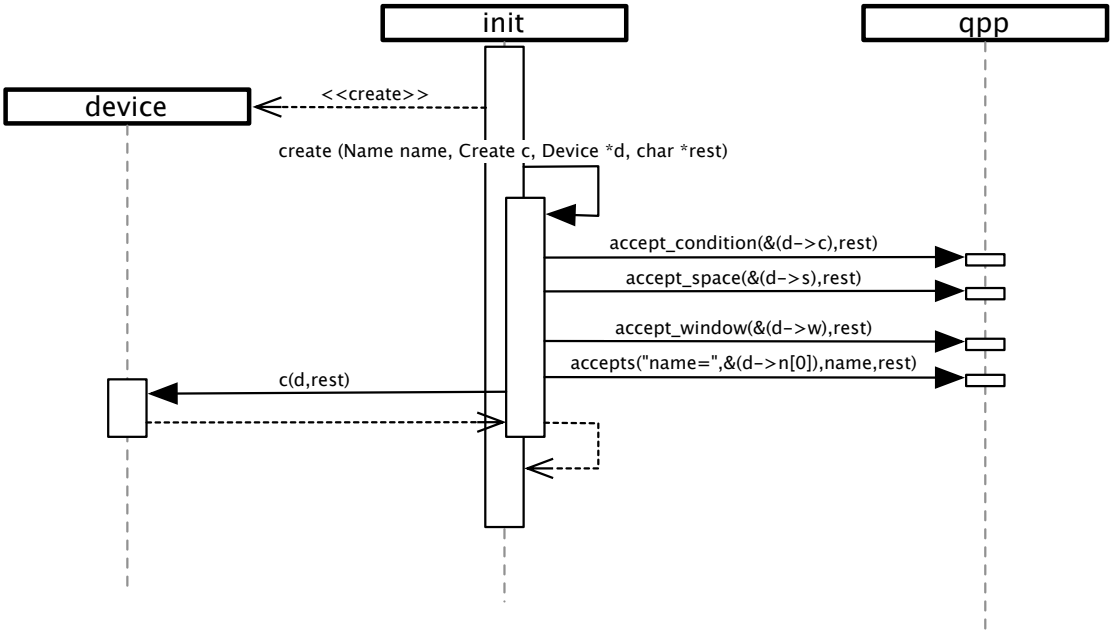


Figure 2.5: UML sequence diagram illustrating device creation. Generic device creation is handled by `init`, using `qpp` functions. Device-specific initialisation is performed by the device code.

arguments than full calls to QPP functions. The macros, along with details of their arguments and function, are listed in Appendix B. Each macro uses a QPP function to search the calling device's parameter string, `w`, for a key name — such as "cheese" in `cheese=3.65` — and parses the portion after the assignment operator according to the datatype required by that particular macro. If the type matches and any additional constraints are met, the resulting value or pointer is stored in a variable of the same name in the calling device's `STR` structure. Where it is not appropriate to specify a default value, it is possible to 'demand' a value of the QUI script by using a null value for the default. Nulls differ dependent on the datatype expected.

It is recommended that device developers employ these macros rather than have devices call functions of *qpp.h* directly. Doing so will improve the legibility of device code. Also, by coding devices to the macro interface, rather than the explicit implementation of QPP functions, device code is more loosely coupled to QPP.

2.4.3 Defining k_variables

As described in Section 2.2.3, variables defined in the script are distinct from the C language variables used in QUI's implementation. For clarity, variables defined in the script, using `def` may be referred to as k_variables. Defining a k_variable adds a new symbol to one of QUI's symbol tables.

Symbol Tables

Symbol tables are the mechanism by which QUI keeps track of macros and variables defined in the script. By default, there are two symbol tables, both in *k_.h*; `systb` for reserved constants and function names, and `deftb` for user-defined string macros and k_variables.

Each symbol in a table represents a macro, variable or function. The table records the name and type of each k_variable along with a pointer to its value. In the case of functions, the symbol table records the function name, number of parameters, return type and a pointer to the function.

When `init()` reads a `def` command, it calls `def()` (*qpp.c*), passing the string that follows `def` keyword. `def()` will insert the variable as a symbol into `deftb`, with the defined type. If the variable's name already exists in either `systb` or `deftb` an error will be raised and QUI will quit. However, since the names of string macros are wrapped in brackets `[ ]`, it is possible for a script to define macros and variables with the same name.

2.4.4 Initialising the `state` Module

The `state` module is responsible for defining the dimensionality of the problem, the size of the mesh and the number of dynamic variables per point. It also allocates the main array, `New`, which stores the state of dynamic variables for each point in the mesh. `New` is made available publicly via the *state.h* header. *state.h* also provides metadata relating to `New` in variables listed in Table 2.4.

Name	Description
`t`	The current time step.
`New`	The simulation medium.
`vmax`	Number of layers.
`xmax`	Number of points in the x axis.
`ymax`	Number of points in the y axis.
`zmax`	Number of points in the z axis.
`vmax_xmax`	Number of variables in the v/x plane.
`vmax_xmax_ymax`	Number of variables in the $v/x/y$ plane.
`DX`	The number of array locations between neighbouring points on the x axis. (See Figure 2.8)
`DY`	The number of array locations between neighbouring points on the y axis. (See Figure 2.8
`DZ`	The number of array locations between neighbouring points on the z axis.
`DV`	The number of array locations between neighbouring points on the v (variable) axis.
`dim`	The dimensionality of the mesh, from `0` (single cell) to `3`.
`ONE`	`1` if the dimensionality is greater than or equal to 1. `0` otherwise.
`TWO`	`1` if the dimensionality is greater than or equal to 2. `0` otherwise.
`TRI`	`1` if the dimensionality is greater than or equal to 3. `0` otherwise.
`X0`	Lower x 'bound of internity' of the mesh.
`X1`	Upper x 'bound of internity' of the mesh.
`Y0`	Lower y 'bound of internity' of the mesh.
`Y1`	Upper y 'bound of internity' of the mesh.
`Z0`	Lower z 'bound of internity' of the mesh.
`Z1`	Upper z 'bound of internity' of the mesh.

Table 2.4: Public `state` Variables

When `init()` reads the `state` command, it calls the `state()` function (*state.c*) to read the mesh dimensions from the parameter string of the `state` command and allocate memory for `New`

accordingly. `init()` requires that `state` is called before any devices and that `state` is called no more than once. The process followed by `state()` is illustrated in Figure 2.6.

Bounds of Internity

The 'bounds of internity', `X0`, `X1`, `Y0`, `Y1`, `Z0` and `Z1` (*state.h*), listed in Table 2.4, relate to the edges of the 'internal' medium. `xmax`, `ymax` and `zmax` define the number of points in the mesh along each axis. An active axis is one with length, i.e. number of points, ≥ 3. One point at the extreme of each *active* axis is considered, and will be used as, a boundary point. The bounds of internity mark the first and last points between the boundary points. This is shown in Figure 2.7. Active axes are counted in `state()` (*state.c*) to decide the dimensionality of the medium, `dim`. The variables `ONE`, `TWO` and `TRI` are used as 'flags' to indicate if the mesh is at least one-, two- or three-dimensional, respectively. It is possible for QUI to run a zero-dimensional simulation, in which the mesh will consist of a single point/cell.

`xmax`, `ymax`, `zmax` and `vmax`, along with the limits of internity, are defined as k_variables of type `real` for reference from within the script.

The `New` Array

`New` is a one-dimensional array, representing four-dimensional data, i.e. multiple dynamic variables — referred to by QUI as 'layers' — at locations in three-dimensional space. Figure 2.8 shows how locations in `New` relate to the states of four cells, each with two layers in a two-Dimensional space of 2×2 cells. Coordinates $\{x, y\}$ are shown on each cell.

To create the four-dimensional view of `New`, *state.h* defines a function macro, `ind()`, to map $\{x, y, z, v\}$ coordinates to an index of `New`, using the equation shown in (2.1).

$$\text{ind}(x, y, z, v) = (z \times vmax\_xmax\_ymax) + (y \times vmax\_xmax) + (x \times vmax) + v \tag{2.1}$$

In order to identify relative locations in `New`, *state.h* provides four shift macros, `DX`, `DY`, `DZ` and `DV`. For example, given an index of `New`, adding `DY` would give the index of the same variable at the next point along the y axis. Figure 2.8 illustrates how shift macros relate to the indices of `New`.

`New` is the only representation of the simulation state in memory; there is no `Old`. `New` contains one time-slice of the solution. Following the explicit Euler time integration method (1.13), to compute one time-slice, one needs the solution at the previous time step. However, in QUI, there is no `Old` containing the previous timestep solution. Maintaining only one timestep halves the memory necessary to run QUI, but requires caution when computing reading from and writing to the same array. In this setting an operation such as the computation of the Laplacian (1.12) is non-commutative if the function both reads from and writes to the same layer of `New`. It is necessary to break the operation into two parts, storing interim values in an additional layer of `New`. Such an approach is taken by QUI's `diff*` devices, described in Section 2.9, which compute the Laplacian at each timestep and store it in an extra dynamic variable to be used at the next timestep. The RHS module can then read the stored Laplacian value and modify the transmembrane voltage accordingly. Splitting the operation in this way obviates the need for the whole previous time-slice to be held in memory.

2.4.5 Creating the Ring of Devices

The ring of devices that operate on the simulation medium is implemented as an array of `Device` structures, `dev`, declared in *qui.c*. Each device call in the script will populate a `Device` structure

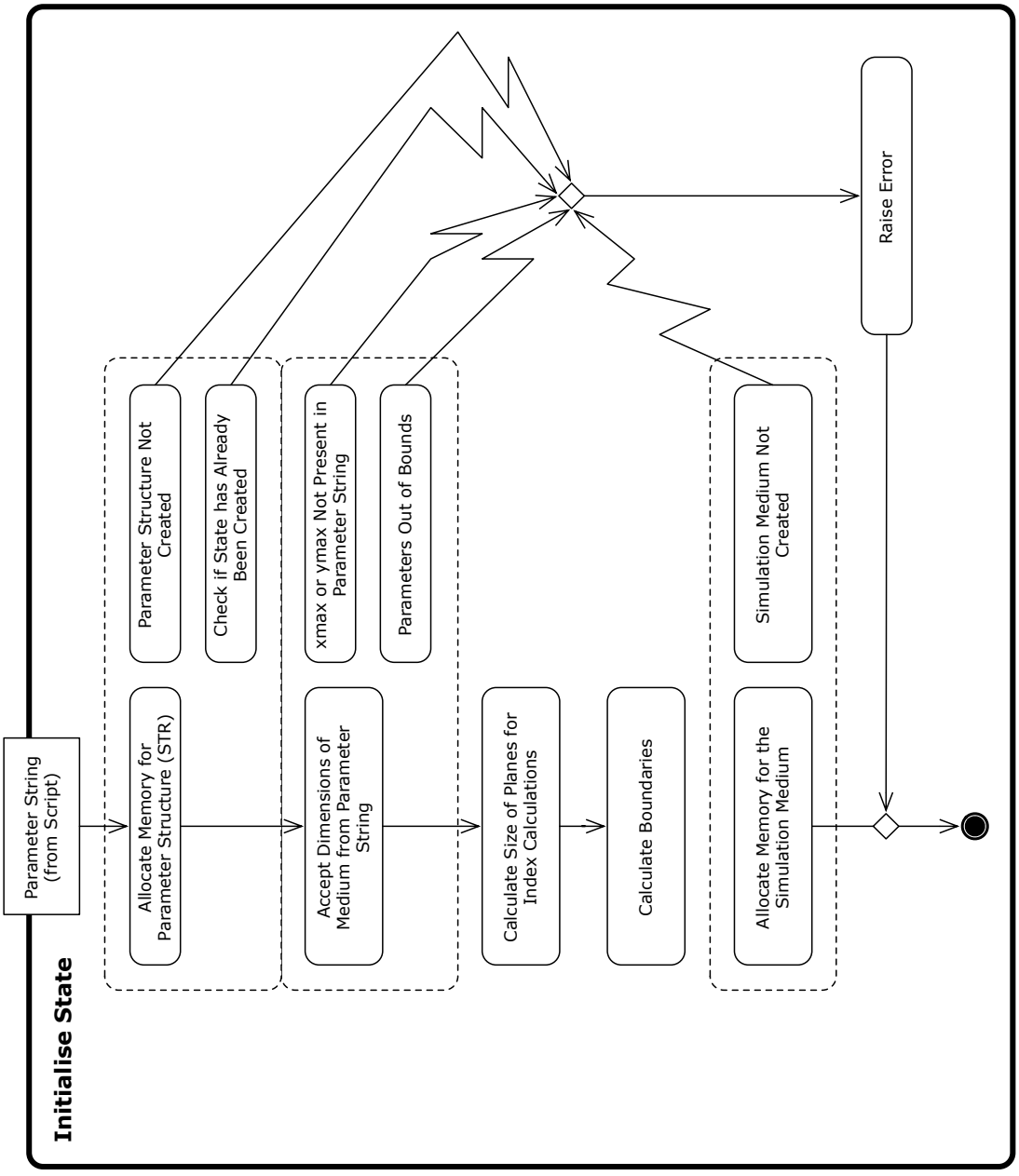


Figure 2.6: UML activity diagram show the initialisation of the `state` module.

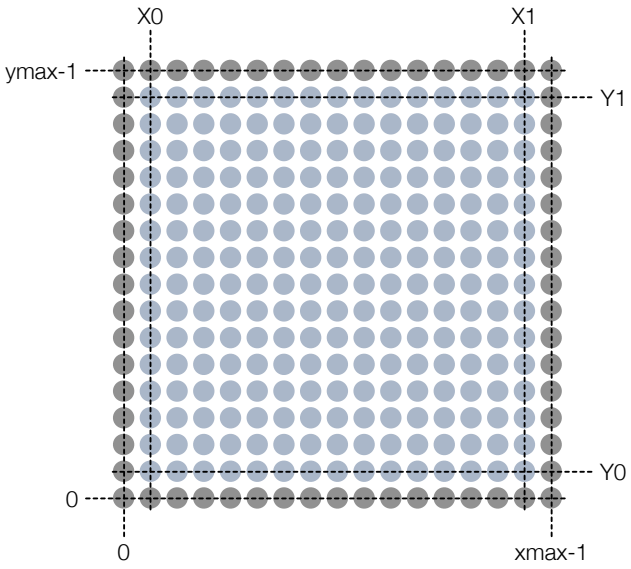


Figure 2.7: Bounds of internity. The variables `X0`, `X1`, `Y0`, `Y1`, `Z0` and `Z1` mark the first and last points of the medium that are *not* boundary points. The active medium is shown in light blue, with boundaries shown in dark blue.

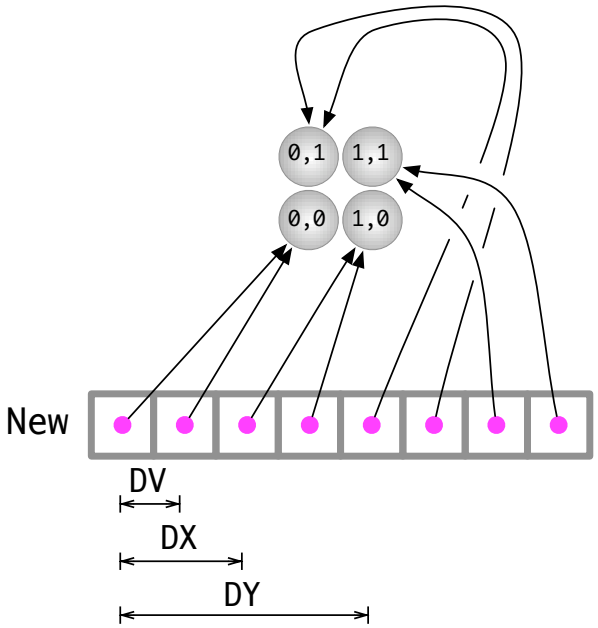


Figure 2.8: Relationship between cartesian coordinates of the simulated medium and array indices of `New`. Shift macros `DV`, `DX` and `DY` show the distance in `New` between neighbouring points on the v, x and y axes respectively.

in the ring of devices from the supplied parameters. Each `Device` structure brings together the relevant parameter values and function pointers for one instance of a device.

When a device call is read by `init()`, it calls `create()` (*init.c*) to create the device. `create()` uses QPP functions `accept_space()`, `accept_name()` and `accept_window()` to read the generic device parameters from the script and populate the corresponding fields of the `Device` structure. Finally, the device's Create function is called, as discussed in Section 2.3.9.

The sequence of calls involved in the above procedure is illustrated in Figure 2.5 and code for the `create()` function can be seen in Listing A.4.

2.5 Running the Simulation

The `main()` function (*qui.c*) is responsible for running the simulation. Listing 2.13 shows the time loop of `main()`. At each timestep, QUI iterates over the ring of devices. If a device's condition evaluates to TRUE, its run function (`d.p()`) will be called. If the device returns `0`, either due to an error, or because it is a `stop` device, QUI will terminate the simulation. This process is illustrated in Figure 2.9.

```
t = 0; // Current timestep
// Timestep loop
for(;;) {
    // Iterate over the ring of devices
    for (idev=0; idev<ndev; idev++) {
        #define d (dev[idev])
        // If the device's condition is true,
        if (*(d.c)) {
            // call the device's run function.
            if (!d.p(d.s, d.w, d.par )) goto Exit;
        }
    }
    t++;
};
```

Listing 2.13: .]*qui.c* — Simulation time loop within `main()` [edited for simplicity].

2.6 Ending the Simulation

When a device's run function returns `0`, the main loop in *qui.c* is broken and control jumps to the `Exit` block:

The `Exit` block reports the name of the device that caused the simulation to stop, along with the current timestep. `term()` (*init.c*) is then called. `term()` iterates through each of the devices in the ring, calling their Destroy functions. Finally the simulation medium and and symbol tables are freed via `state_free()` (*state.c*) and `term_const()` (*qpp.c*). Code for the `term()` function is shown in Listing A.5.

2.6.1 The stop Device

The `stop` device's only purpose is to bring the simulation to an end; its run function will always return `0`. By using the device's `when` parameter, the user can define when the simulation should stop.

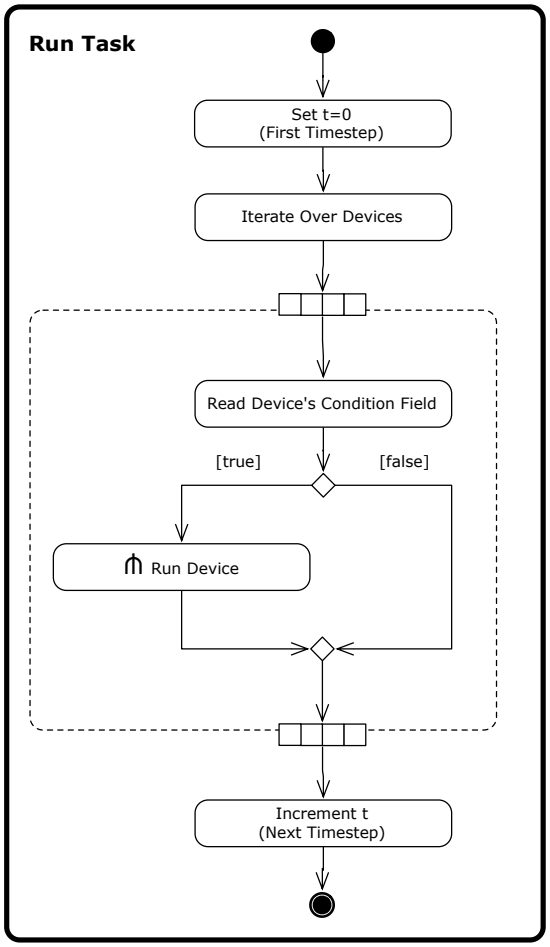


Figure 2.9: UML activity diagram illustrating the running of a QUI simulation.

2.7 The `euler` Device

Defined in *euler.c*, `euler` is a general pointwise ODE stepper used to compute future values of dynamic variables using the *Euler forward method* (1.13). The Euler forward method calculates the value of a variable at the next time step to be the existing value plus the product of the derivative and time step. The calculation takes the form:

$$\mathbf{u}^{t+1} = \mathbf{u}^t + \Delta t \cdot \mathrm{d}\mathbf{u} \tag{2.2}$$

where $\mathbf{u} = (V, m, h, \dots)^T \in \mathbb{R}^n$ is the vector of dynamic variables, defined as the number of dynamic variables of the cell model, and $\mathrm{d}\mathbf{u} = (\mathrm{d}V, \mathrm{d}m, \mathrm{d}h, \dots)$ is the vector of time derivatives corresponding to the variables of $\mathbf{u}$.

Models of kinetics to be used with `euler` are implemented as RHS (Right Hand Side) modules. RHS modules are similar in construction to devices in that they group related data structures and function pointers in a `struct`.

With reference to (2.2), RHS modules may either compute $\mathrm{d}\mathbf{u}$ by providing derivatives for

each dynamic variable of $\mathbf{u}$, or solve $\mathbf{u}^{t+1}$ directly. RHS modules are discussed in detail in Section 2.8.

2.7.1 Creating euler

The euler device's Create function accepts values for three parameters from the QUI script, as detailed in Table 2.5. The RHS device to be used is specified via the ode parameter.

Type	Name	Description
string	ode	The name of the system of Ordinary Differential Equations to use for the Right Hand Sides. This should correspond to the name of an RHS module in *rhslist.h*.
real	ht	The time step duration.
int	rest	Number of time steps required to find resting state.

Table 2.5: euler QUI Script Parameters

Figure 2.11 illustrates the steps involved in the euler device's create function. Figures 2.11 and 2.12 expand on Figure 2.5 to show the calls involved in creating the euler device and an associated RHS module, with Figure 2.12 adding the data-flow of parameters populating fields of the device and RHS parameter structures.

The euler Device's Parameter Structure

Like all devices, euler has a parameter structure to store persistent data for its state. The euler device's parameter structure, defined in *rhs.h* as rhs_str, is detailed in Table 2.6. Crucially, it contains pointers to the RhsProc function of the RHS module (equivalent to a device's run function) and to the RHS device's STR structure.

Type	Name	Description
real	ht	Time step duration, in ms
RhsProc	*f	Pointer to the RHS module's run function.
real	*u	Pointer to the location in New of the current cell's state.
real	*du	Pointer to the vector of derivatives.
Par	p	Parameters of the RHS module.
Name	ode	Name of the ODE system.
int	rest	Number of time steps required to find resting values.
Var	var	Description of dependent parameters.

Table 2.6: Fields of the euler parameter structure.

Many cell models will, without external stimulation, return to a stable 'resting' state after some time. To begin a simulation with a rested medium, it may be necessary to run the RHS module for some time until its resting state. To facilitate this, euler provides the rest parameter, that allows the user to specify the number of timesteps for which an RHS function should be run before starting the simulation. The number of time steps taken to find a resting state is user-defined via rest. euler makes no attempt to measure the stability of any resting state.

To improve the efficiency of this operation, euler runs the RHS function of a single point rest times, before copying the resulting state to all points in the mesh. The values of dynamic

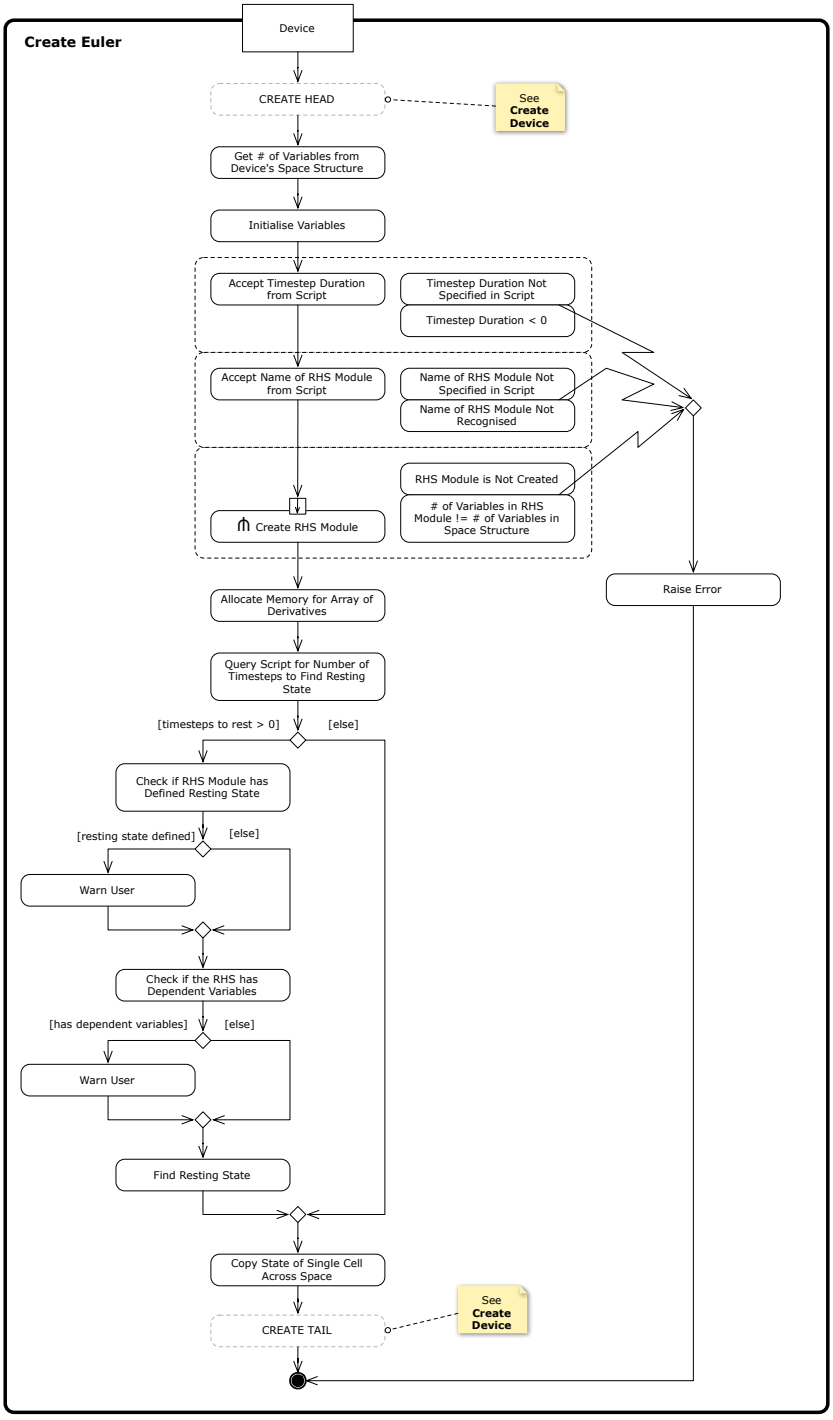


Figure 2.10: UML activity diagram showing creation of the `euler` device.

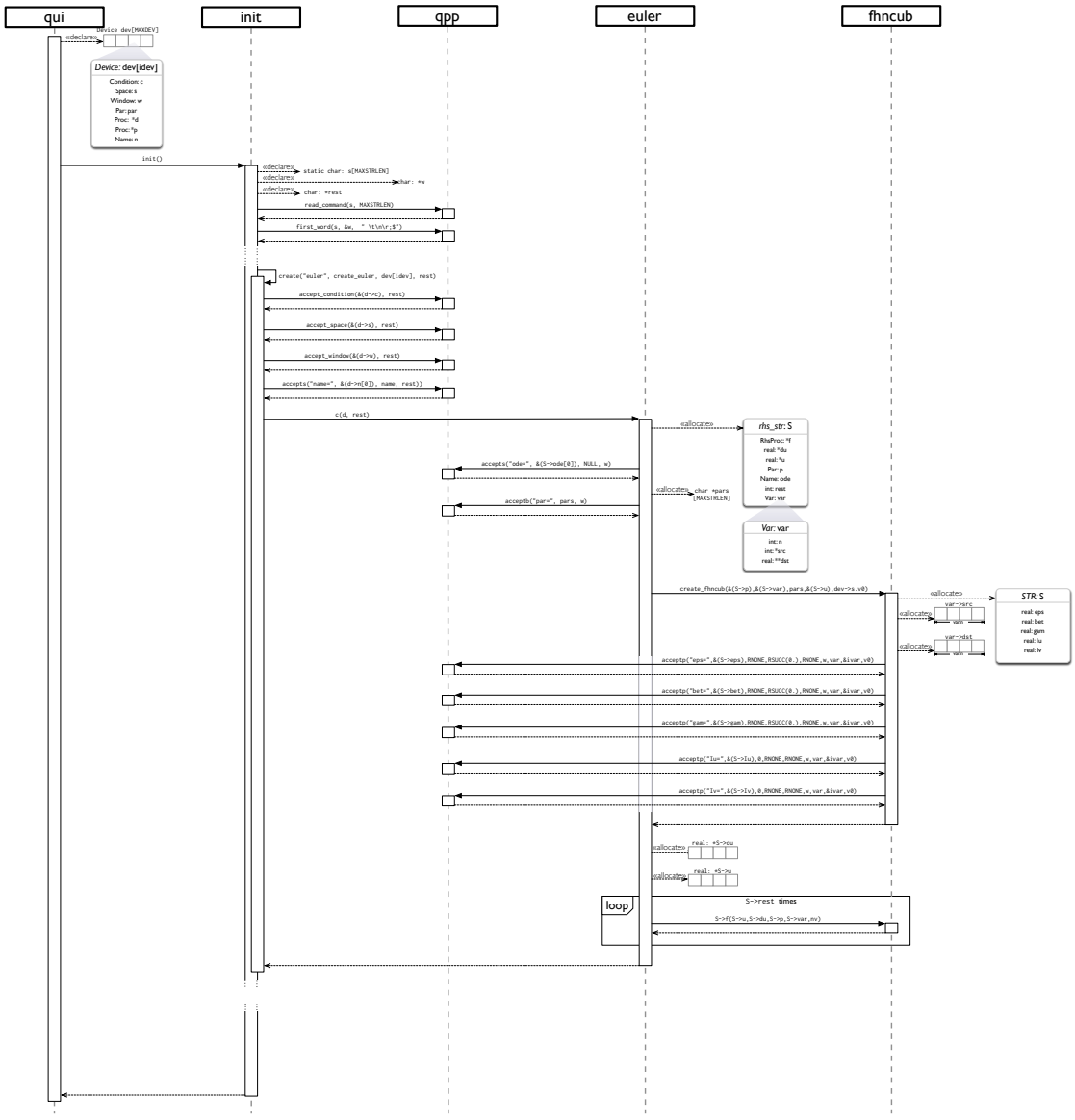


Figure 2.11: UML sequence diagram illustrating the process of creating the `euler` device, including its RHS module, `fhncub`. A larger version of this figure is available on the CD accompanying the printed edition of this thesis.

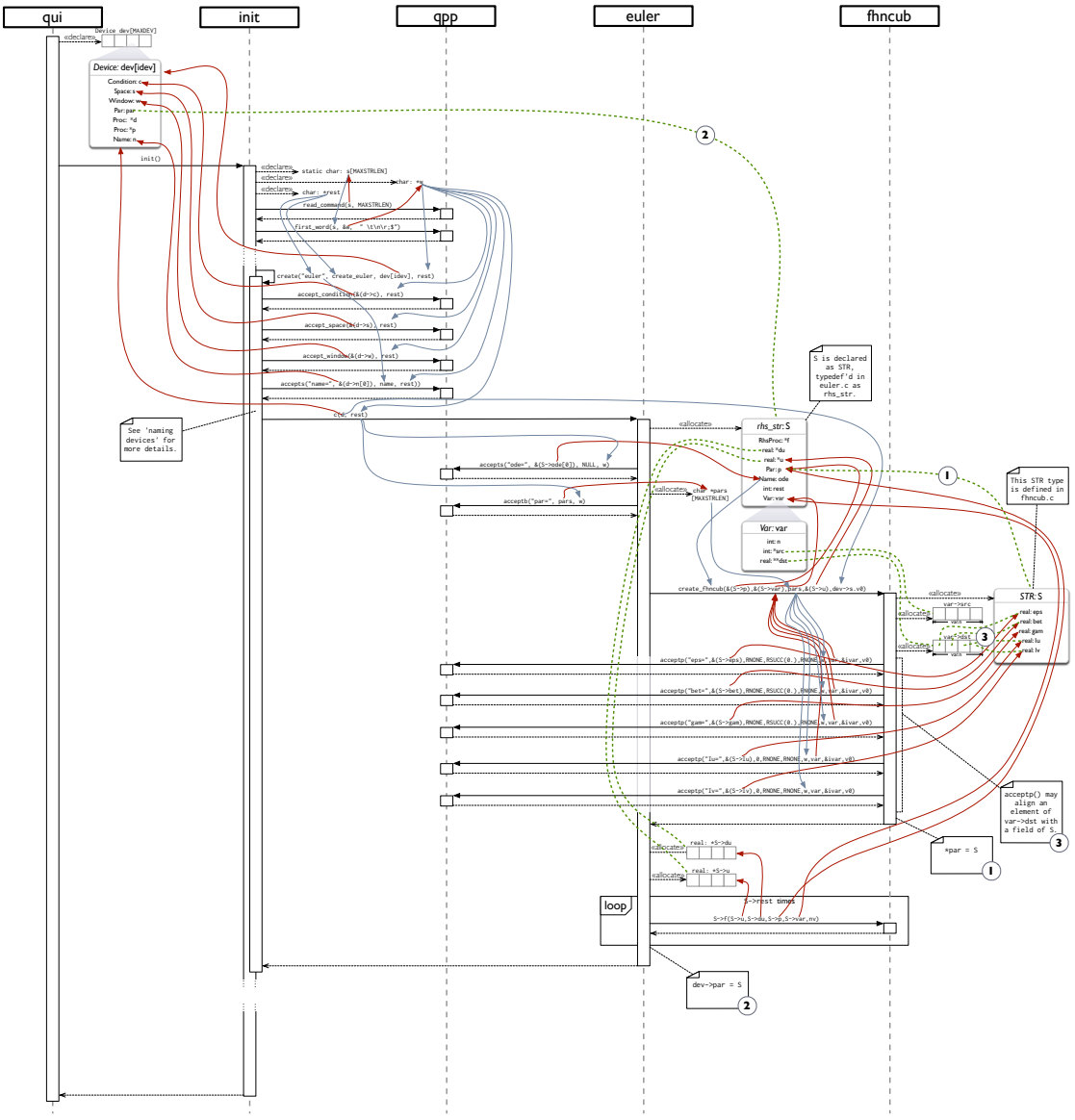


Figure 2.12: UML sequence diagram illustrating the process of creating the `euler` device, including its RHS module, `fhncub`. Arrows indicate the flow of parameters. Blue arrows show input parameters; red arrows show output parameters. Dashed green lines indicate aligned pointers. A larger version of this figure is available on the CD accompanying the printed edition of this thesis.

variables arrived at are then duplicated across all of the cells in the medium. An alternative to calculating a resting state in `euler` is for the RHS module to provide resting values to its dynamic variables inside its `create` function. In the event that a value is given for `rest` when initial values are provided, `euler` will output a warning. The `euler` device's code for setting initial conditions is given in Listing A.6.

2.7.2 Running Euler

The `euler` device's Run function retrieves values for the following parameters from its parameter structure:

- The time step duration, `ht`.
- The RhsProc function of the RHS module.
- The RHS module's parameter structure.
- The vector of derivatives, `du`.
- The name of the ODE system, used for error messages.
- The RHS module's description of its dependent parameters (discussed in Section 2.8.3).

`euler` then iterates through each point in its space, updating an array pointer, `u`, which it then passes to the RHS module's `RhsProc` function to compute $\mathbf{u}^{t+1}$ at that location. RHS modules may either update the values in `u` directly, or update a vector of derivatives, `du`, and let `euler` update values in `u` using (2.2). This process is shown in Listing A.7 and illustrated in Figure 2.13.

2.8 Right Hand Side Modules

Right Hand Side (RHS) modules compute the kinetics, or ionic currents in physiologically detailed models, of the reaction-diffusion equations (1.10); the 'Right Hand Sides' of the ordinary differential equations used to update the state of individual cells (e.g. 1.1).

RHS modules utilise functionality provided by macros in the header *rhs.h*, similar to device template macros described in Section 2.3.8. RHS modules also define a parameter structure, of type `STR`, and functions to create and run the RHS module. It is important to note, however, that RHS modules do not use the `Device` datatype, nor do they form part of the 'ring of devices' directly, instead they operate as sub-components of the `euler` device. As such, the term 'module' is used to distinguish them from devices proper. To prevent confusion with devices, RHS modules are said to have RHSCreate and RHSProc functions, as opposed to Create and Run functions.

rhs.h provides template macros for RHSCreate (`RHS_CREATE_HEAD` and `RHS_CREATE_TAIL`) and RHSProc (`RHS_HEAD` and `RHS_TAIL`) functions. These are discussed in Sections 2.8.4 and 2.8.5, respectively.

2.8.1 The RHS Parameter Structure

Like a device, each RHS module has a parameter structure of type `STR`. As for devices, `STR` is a 'marker' type, with no default definition. Each RHS module defines its own `STR` type to suit its own needs. By making parameters of the module available globally through the device structure, RHS modules can economise on the number of dynamic variables that must be stored in `New`. The allocation of the RHS module's parameter structure, and its subsequent assignment

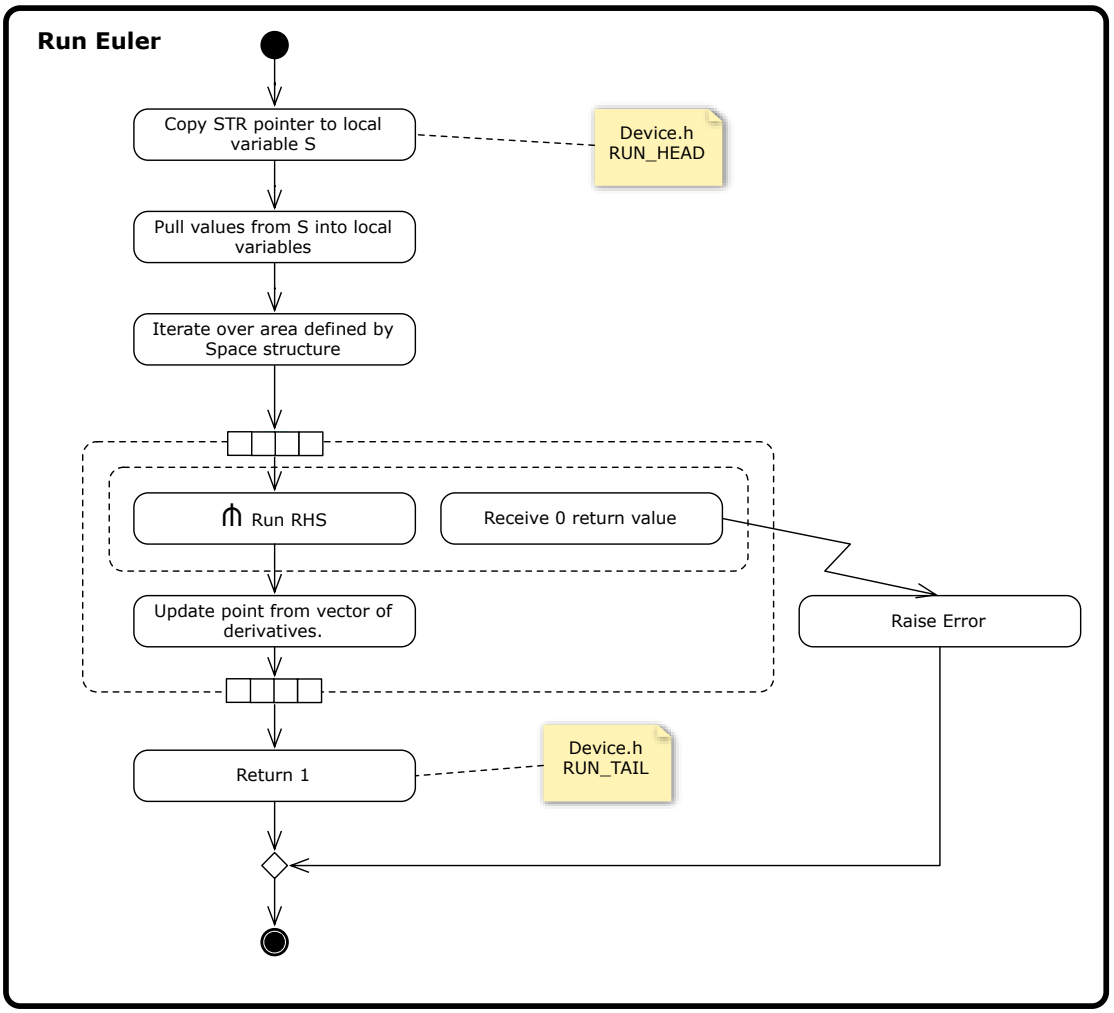


Figure 2.13: UML activity diagram showing the running of the `euler` device. Execution of the ODE is delegated to the RHS module.

to the `p` field of the `euler` device's parameter structure, is described in Section 2.8.4. Within RHS module code, the parameter structure is referred to as `S`.

2.8.2 `u`

The first parameter of the RHSProc function, `u`, is a reference to the first dynamic variable of the RHS module in `New` corresponding to the point in space on which the `euler` device is currently operating. To improve legibility of the code, RHS modules commonly employ `enum` structures to associate names with indices of `u`. `u` should contain only the dynamic variables of the model of kinetics. To avoid redundancy, all model parameters should be stored in its parameter structure. All dynamic variables of the RHS module will be updated by the `euler` device in the manner described in Section 2.7.2.

2.8.3 Dependent Parameters

In addition to parameters and dynamic variables of the RHS module, a simulation may require the use of space-dependent parameters, i.e. a parameter with discrete values at each point in the mesh. The `euler` device facilitates this with a feature known as dependent parameters. A dependent parameter is a layer of `u` that is made available to the RHS module as if defined in the device's parameter structure.

One example of how dependent parameters could be used is to introduce inhomogeneity. For example, as in Tusscher and Panfilov [2003], current density could be varied across the mesh by altering the value of a current's maximum conductance. To achieve this, the user would have to declare an additional layer for `New` by adding 1 to `vmax`. In the call to `euler`, the user would then associate this extra layer with the required parameter of the RHS module, e.g. `gkp`. The user could then use a `k_func` device to set the value of `gkp` at each point in the mesh. When run, the RHS module would use the value from the extra layer in `New`, in place of a fixed, global value.

Dependent parameters are accepted by the RHS module using the `ACCEPTP` macro (*qpp.h*). If the value supplied is a normal script expression, e.g. `1.35` or `foo/4`, the parameter will be assigned a constant value, in the same manner as a parameter accepted with `ACCEPTR`. If the parameter value is of the form `@<layer>`, then the parameter will be associated with layer `<layer>` of `New`. The RHS module's `Var` structure will be used to hold both the value of `<layer>` and a pointer to the parameter in the RHS module's parameter structure. When the RHS module is run, the value of the selected layer will be copied from `u` to the module's parameter structure, so that any future references to the parameter will use the value from `u`.

Var

The definition of the `Var` datatype is shown in Table 2.7.

Type	Name	Description
`int`	`n`	Count of dependent parameters.
`int *`	`src`	Layers of dependent parameters in `u`.
`real **`	`dst`	Pointers to fields of dependent parameters in RHS parameter structure.

Table 2.7: Fields of the `Var` structure.

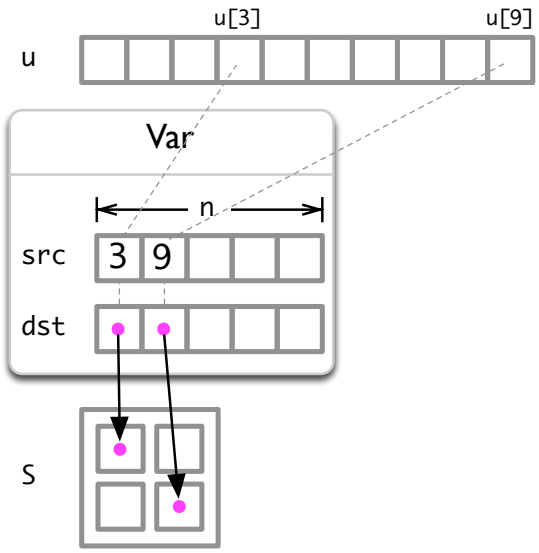


Figure 2.14: Data structures involved in the creation and use of dependent parameters. The `var` datatype associates the values held in `u`, the current position of `New`, with the parameter structure `S`.

Recall that `New` represents a three-dimensional mesh, with `vmax` layers at each point in space. To make use of dependent parameters, users must allocate additional layers in `New` to those required by the RHS module. For example, when using the `fhncub` RHS module, which has two dynamic variables, with a dependent variable for external current, the user should specify `vmax=3`.

The `Var` structure implements the connection between fields of the RHS module's parameter structure and specified layers of `New`. The `Var` structure has 3 fields; `n`, `src` and `dst`. The `n` field is the count of dependent parameters. The `src` and `dst` fields are both pointers to arrays of size `n`. The `src` array contains the layer number in `New` of each dependent variable. For example, if the user specifies `IV=@24` in the script as the first dependent parameter, `src[0]` will equal `24`. Figure 2.14 describes these relationships pictorially.

The `dst` array contains pointers to the dependent parameter in `S`, the RHS module's parameter structure. Elements at the same index of `src` and `dst` relate to the same dependent parameter in `u` and `S` respectively. So, for the parameter assignment above, `dst[0]` points to `S->IV`. By copying source variable to destination parameter, the value of `S->IV` is taken from `u[24]`.The `Var` structure is updated with the locations of dependent parameters when the `RhsCreate` function is run.

2.8.4 Creating an RHS Module

The `RHS_CREATE_HEAD` and `RHS_CREATE_TAIL` template macro define the `RhsCreate` function. Once macros have been expanded, the name of this function becomes `create_<ode>`, where `<ode>` is the name of the RHS module. The `RHS_CREATE_HEAD` macro expands to allocate memory for the RHS module's parameter structure, and assign its address to the local pointer, `S`.

Since the RHS module is not part of the ring of devices, it does not receive an explicit call from the QUI script. To facilitate the scripting of RHS modules, `euler` acts as a conduit, passing

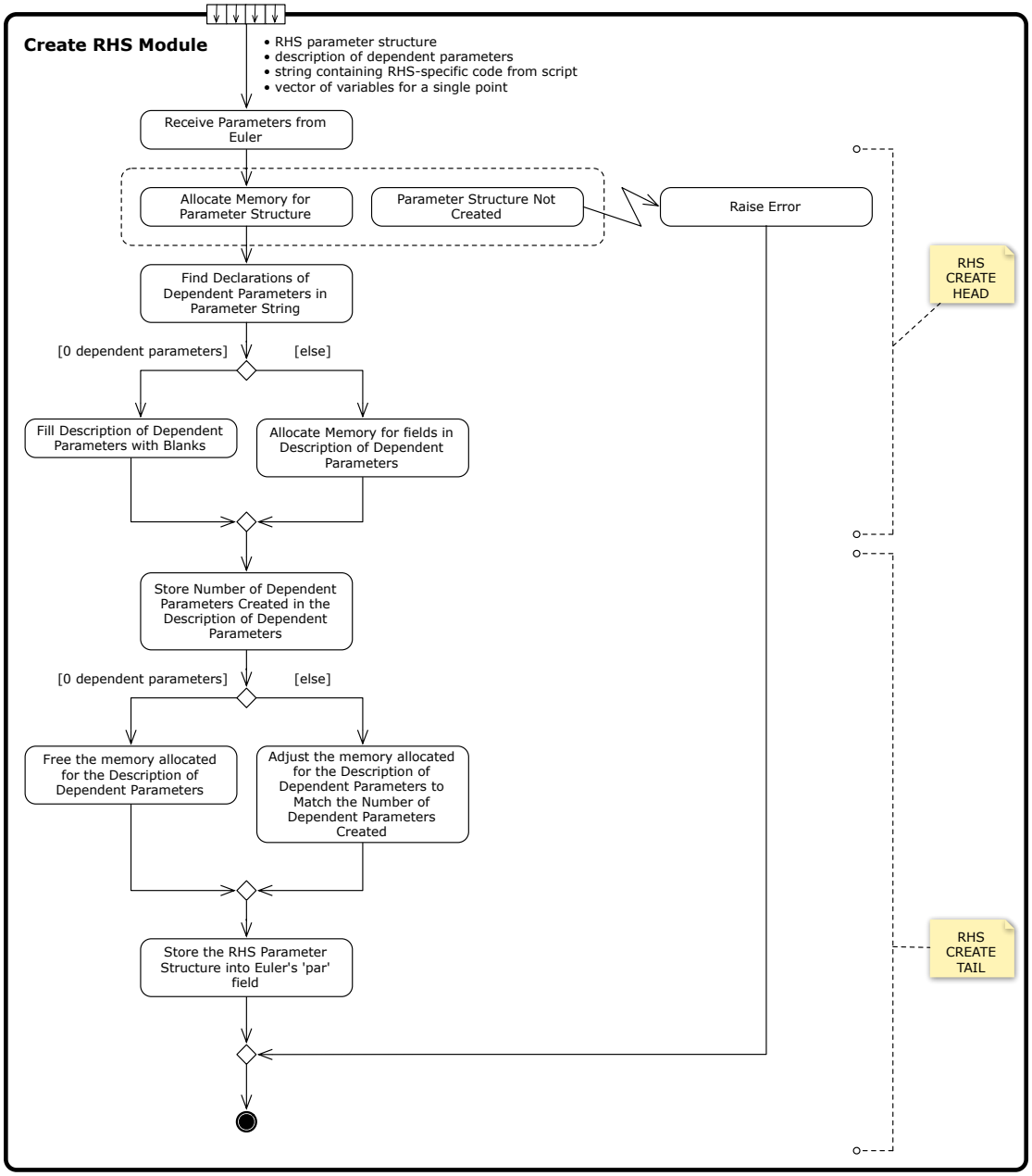


Figure 2.15: UML activity diagram showing creation of an RHS module. Notes show the behaviour defined in the `RHS_CREATE_HEAD` and `RHS_CREATE_TAIL` macros.

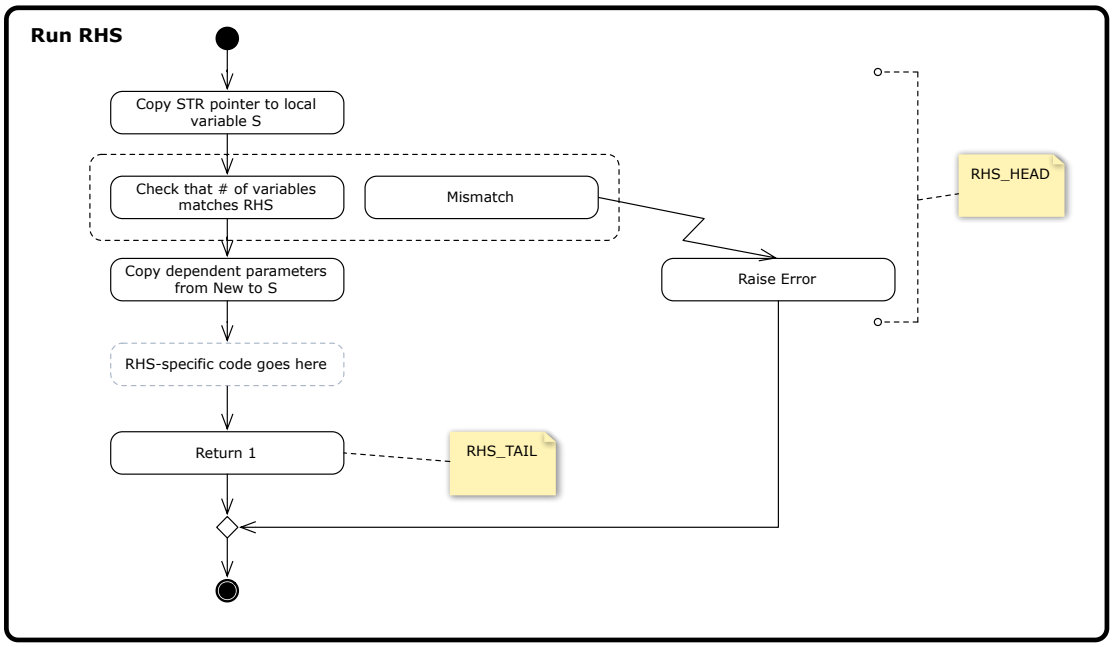


Figure 2.16: UML activity diagram showing the running of an RHS module. Notes show the common behaviour defined by the `RHS_HEAD` and `RHS_TAIL` macros.

script parameters using a block parameter, `par`:

```
euler v0=0 v1=neqn-1 ht=ht ode=lrd par={ht=ht; IV=@[iext]};
```

Listing 2.14: Passing parameters to the RHS module via the `euler` device's `par` parameter.

Any script code contained in the braces following the `par` key is passed to the RHS module as an argument to the `RhsCreate` function. The parameter string is then examined for '`@`' symbols. Each '`@`' in the string indicates a dependent variable. These are counted, and used to allocate space in `Var`'s `dst` and `src` fields. Code for this operation is shown in Listing A.8.

Module-specific initialisation code can accept (optionally dependent) parameters using the `ACCEPTP` macro (*qpp.h*).

The `RHS_CREATE_TAIL` macro readjusts, if necessary, the number of dynamic variables and the memory assigned to `src` and `dst` in `Var`, in response to any additional dependent variables being added by the RHS module in the function body. Finally, the RHS module's `par` field is pointed to the newly initialised `S`, thus assigning the RHS module's parameter structure to the `p` field of the calling `euler` device's parameter structure.

2.8.5 Running an RHS Module

The RHS module is run by calling its `RhsProc` function with the following arguments:

`*u` A pointer to the dynamic variables at the current location in `New`.

`*du` Pointer to the vector of derivatives. Used to update variables in memory using the Euler forward method.

`par` The RHS module's parameter structure.

`var` Description of dependent parameters.

`ln` The number of dynamic variables expected by `euler`. This must match the number of dynamic variables declared by the RHS module.

The `RHS_HEAD` macro expands to assign `par` to local variable `S`. It then checks that the number of dynamic variables declared by the RHS modules matches that of the caller (`euler`), exiting the program if not. This ensures the safety of `euler`'s iterations over the variables in `u`.

If there are any dependent variables, values of the dependent variables are copied from `u` to the corresponding field of `S`, as shown in Listing A.9.

Following the module-specific code, the `RHS_TAIL` macro inserts a return value of 1, which is used to check that the function ran successfully, and a closing brace.

2.9 Diffusion Devices

QUI provides devices to model isotropic diffusion. QUI's diffusion devices compute the Laplacian $D\nabla^2\mathbf{u}$, where D is the scalar diffusion coefficient. For two dimensional simulations in Cartesian coordinates it takes the form

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}. \tag{2.3}$$

The `diff1d`, `diff2d`, and `diff3d` modules calculate the diffusion of voltage in one, two, and three dimensions respectively.

As befits the majority of cardiac simulations, QUI's diffusion devices operate on transmembrane voltage only. The devices use the `v0` space parameter to indicate the layer from which transmembrane voltage should be read. The computed Laplacian is assigned to the layer specified by `v1`. For a one-dimensional medium, the Laplacian is calculated as follows:

$$\nabla^2\mathbf{u} = \frac{V_{x-1} + V_{x+1} - 2V_x}{\Delta^2} \tag{2.4}$$

where $\nabla^2\mathbf{u}$ is the Laplacian, Δ is the Laplacian space step, and V is the transmembrane voltage of a point, with the subscript $x-1$ indicating the left neighbour of cell x, and $x+1$ indicating its right neighbour. The `diff1d` device's implementation of (2.4) is shown in Listing A.10.

The Laplacian value is often scaled using a diffusion coefficient, D. The diffusion coefficient has no direct anatomical analog, but can be used to 'tune' simulations so as to produce conduction velocities that match experimental observations. The effect of the diffusion coefficient is to scale waves and, as such, the constant can also be used to 'fit' phenomena within a given simulation window, discretisation permitting.

Since the Laplacian value from the *previous* timestep is used by the RHS module, the value should be protected from changes by declaring it as a dependent parameter. See Section 2.8.3 for more information on dependent parameters.

2.10 Devices for Setting Boundary Conditions

QUI provides three suites of devices for boundary conditions. Devices with names beginning `dirich` compute Dirichlet boundary conditions, devices beginning `neum` compute Neumann

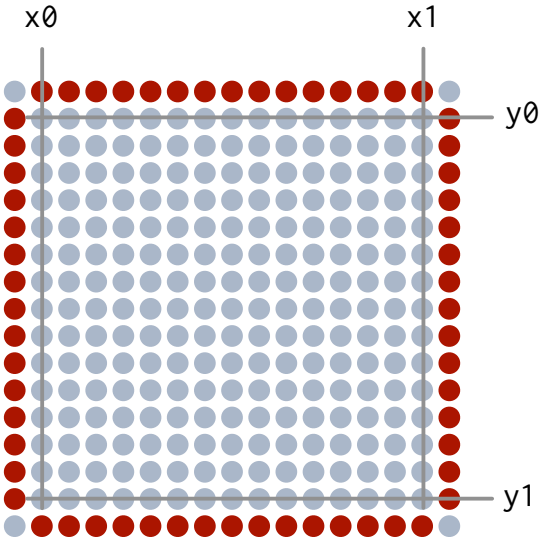


Figure 2.17: Effect of `dirich` devices relative to their space. Values are written to red points.

boundary conditions and devices beginning `tor` compute torus conditions. For each type of boundary condition there are five devices; one for each axis of the mesh — e.g. `dirichx` or `neumy` — plus 2D and 3D devices incorporating (x, y) and (x, y, z) respectively, such as `neum2d` or `tor3d`.

Ordinarily, boundary conditions devices operate on the default space. Values computed by boundary conditions devices are assigned immediately *outside* of the device's space, e.g. `neumx` will iterate over the y and z axes, assigning values to points at `0` and `xmax-1`. Points at which values are assigned by boundary conditions devices are referred to here as boundary points.

2.10.1 Dirichlet Boundary Devices

`dirich` devices assign a fixed value to boundary points. They accept a single device-specific parameter, `value` and assign that value to all dynamic variables immediately (and orthogonally — corners are not updated) outside the device space. Points affected are shown in Figure 2.17.

2.10.2 Neumann Boundary Devices

As shown in Figure 2.18, `neum` devices assign the value from the last point within the device space to its neighbouring boundary point.

2.10.3 Toroidal Boundary Devices

`tor` devices create the effect of a toroidal mesh, which has no edges. This is achieved by duplicating all dynamic variables of the last points inside the space to boundary points at the opposite extreme. For example, the values of all dynamic variables at (`x1`,1,0) will be duplicated to those at (`x0-1`,1,0). This is illustrated in Figure 2.19. For clarity, only the computation of the x axis boundaries is shown.

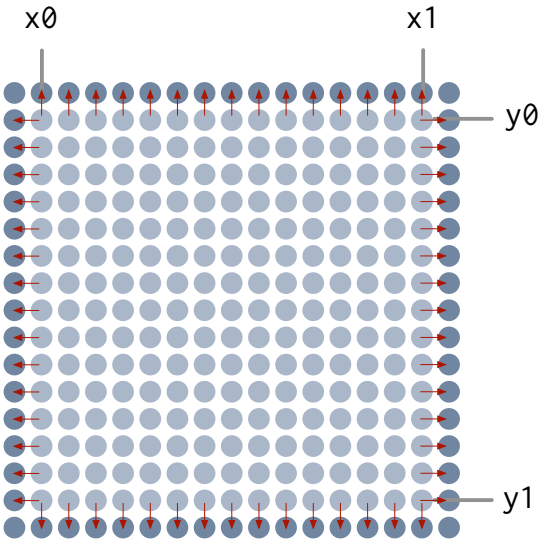


Figure 2.18: Effect of `neum` devices relative to their space. Values are copied from light blue points at the extremes of the space along red arrows to dark blue points immediately outside the space.

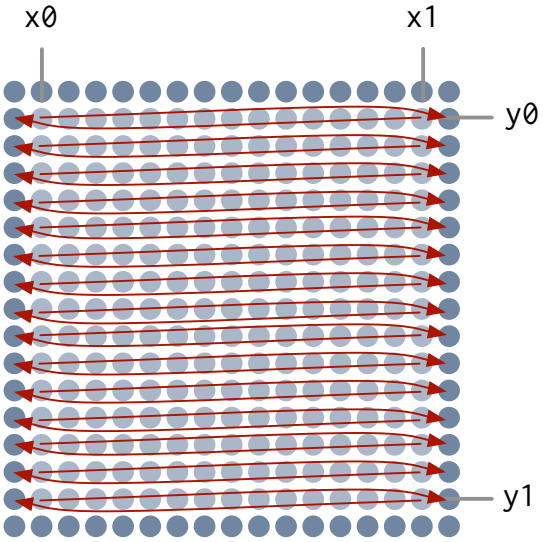


Figure 2.19: Effect of `tor` devices relative to their space. Values are copied from light blue points at the extremes of the space along red arrows to dark blue points at the opposite extreme. For clarity, only x-axis copies are shown.

2.11 The k_func Device

2.11.1 Overview

The k_func device, defined in *k_func.c*, allows the user to specify statements of QUI script code to be executed as part of the simulation. Script code run by a k_func device has access to variables of the simulation medium. The script code run by a k_func device is referred to as its *program*. The program is defined as a code block to the device's pgm parameter:

A common application of k_func devices is in updating variables used for devices' when parameters. For example, to run another device at timestep 200 only, the k_func device called below will assign the value 1.0 to stim when the current timestep, t equals stimTime:

```
def real stimTime 200;
k_func [nowhere] pgm={
  stim = eq(t,stimTime);
};
```

The pgm parameter takes script code between curly braces ({ }). There may be multiple lines, each terminated by a semicolon. When run, k_func iterates over the area of the medium defined by its Space structure, executing the program at each point, with data from that location of New.

k_func also provides a phase distribution facility by which the contents of a data file may be mapped to dynamic variables.

2.11.2 The k_func Parameter Structure

Fields of the k_func device's parameter structure are described in Table 2.8. Code defining the k_func device's parameter structure is shown in Listing A.11.

2.11.3 Creating k_func

The k_func device's Create function begins by allocating the S->u array to hold the values of all dynamic variables of one point in the mesh.

Next, the file parameter is read. If specified, the corresponding file handle will be assigned to S->file. The file is assumed to contain values of real number variables in tabular form, with rows separated by line breaks and variables separated by any of the delimiters, "\/,; rtn".

For example, the following data file contains a 2×10 table of values:

```
2.378 5.111
5.768 8.860
5.609 7.777
1.493 6.545
5.609 7.777
1.347 7.346
4.256 7.423
4.167 6.245
1.234 7.987
3.245 5.222
```

Listing 2.15: Sample k_func file.

Type	Name	Description
int	ncode	Number of statements in program.
pp_fn	code	Array of size ncode, of pointers to pointers, to the right hand side of each statement in the program.
p_real	data	Array of size ncode, of pointers to the variables on the left hand side of each statement in the program.
double *	u	Array of size vmax, used to hold the values of all dynamic variables of one point in the mesh.
double	x	x coordinate of current point in the mesh.
double	y	y coordinate of current point in the mesh.
double	z	z coordinate of current point in the mesh.
int	np	Number of lines in the file specified by the file parameter, if specified.
int	nv	Number of columns, separated by delimiters, in the first line of the file, if specified.
real *	a	Array, of size np*nv, used to hold values read from the file, if specified.
double *	p	Array, of size vmax, used to hold the results of phase distribution with phaseu.
double	phaseu	Used to indicate the phase for phase distribution to u.
double	phasep	Used to indicate the phase for phase distribution to p.
char[]	filename	Name of the file to be read from, if specified.
FILE *	file	File handle for the file to be read from, if specified.
char[]	debugname	Name of the file to write debug information to, if specified via the debug parameter.
FILE *	debug	File handle of the file to write debug information to, if specified via the debug parameter.
p_tb	loctb	Local copy of the user symbol table, deftb.

Table 2.8: Fields of the k_func device's parameter structure.

The number of variables in the first line is counted and assigned to nv. The number of lines is then counted and assigned to np. Two arrays are then allocated; array to hold data read from the file and pv, to be used for phase distribution, described below.

Each of the values in the file is then read into array[(ip*nv)+iv], where ip is the current line and iv the current variable. In the event that some lines of the file contain differing numbers of variables, extra variables are ignored and absent variables are given the corresponding value from the previous line.

If the first line of the file cannot be read, an error will be raised and QUI will quit. If the file contains only one line, k_func will later ignore phaseu and phasep values, discussed below. Values are assigned to fields of S as shown in Listing A.12. If the debug parameter is specified, the filename and file handle are assigned to S->debugname and S->debug respectively.

k_func then duplicates the entire user symbol table, deftb to a local equivalent, loctb. Since symbol tables associate names with memory locations, the symbols in loctb will have the same values as those in deftb throughout the simulation. The creation of a new table ensures that the user script will not be able to reference the variables inserted by k_func.

`k_func` inserts each of the variables listed below into `loctb`, each aligned with a different field of the parameter structure, `S`.

`x` x coordinate of current location in the mesh. Aligned with `S->x`.

`y` y coordinate of current location in the mesh. Aligned with `S->y`.

`z` z coordinate of current location in the mesh. Aligned with `S->z`.

`u0`–`u(vmax-1)` Variable layers at the current location. Aligned with locations of `S->u`.

If the `file` parameter is specified:

`phaseu` Used for phase distribution. See below. Aligned with `S->phaseu`.

`phasep` Used for phase distribution. See below. Aligned with `S->phasep`.

`p0`–`p(vmax-1)` Used with `phasep`. Aligned with locations of `S->p`.

Finally, the program accepts the `pgm` parameter as a code block before including *k_comp.h*.

k_comp.h, shown in Listing A.13, splits each statement in the program at the assignment operator (`=`). The address of the variable on the left hand side is assigned to a field of `S->data`, while the expression on the right hand side of the operator is compiled into virtual instructions using `compile()` (*k_comp.c*) and assigned to the corresponding field of `S->code`. Virtual instructions are a minimal representation of script code that take the form of a sequence of pointers to functions and their corresponding arguments. These present an efficient way to execute script code as part of the simulation.

2.11.4 Running `k_func`

The `k_func` device's Run function iterates over points of the mesh within the bounds defined by its `Space` structure, using `S->x`, `S->y` and `S->z` as loop indices. Since `x`, `y` and `z` variables in the program are compiled against their namesakes in `S`, their value will always be the current location in the mesh. In each iteration, each of the variables layers at that location are copied to `u`.

Each of the virtual instructions in `S->code` is then executed by the `execute()` function (*k_exec.c*). The result of `execute()` is copied to the corresponding location of `S->data`, thereby assigning the value of the expression on the right hand side to the variable on the left hand side. Where the virtual instruction being executed assigns a value to `phaseu` or `phasep`, phase distribution is initiated.

Conceptually the phase distribution process projects groupings of `nv` locations of `array` — equivalent to one line of the file — onto a circle, 2π in circumference. This is illustrated in Figure 2.20.

The value assigned to `phaseu` or `phasep`, taken to be in radians, indicates a point on the circle, and consequently the line to be read. Since `phaseu` and `phasep` are real numbers, linear interpolation is used to calculate values from two neighbouring rows of `array`.

In the following example, we'll look at `phaseu`. Instructions assigning a value to `phasep` will have an identical effect, except that resulting values will be assigned to locations of `p` instead of `u`.

Phase distribution calculates the value for `phaseu` using `execute()` and then multiplies the value by `np`/2π;. If `phaseu` is negative, its value is increased in steps of `np` until it is positive. Similarly, if `phaseu` is greater than or equal to `np`, its value is reduced in steps of `np` until it is less than `np`.

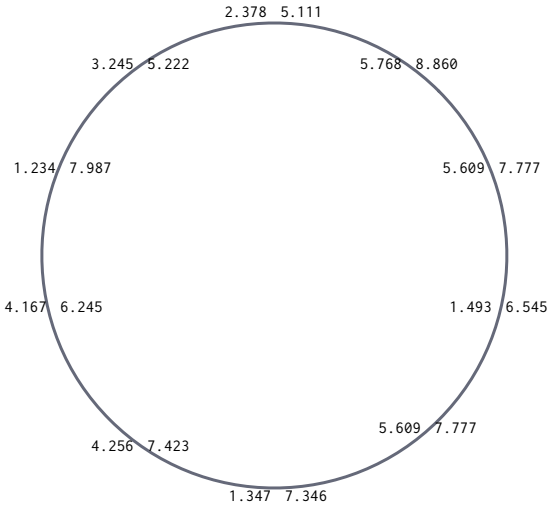


Figure 2.20: Conceptual arrangement of file for use with phase distribution.

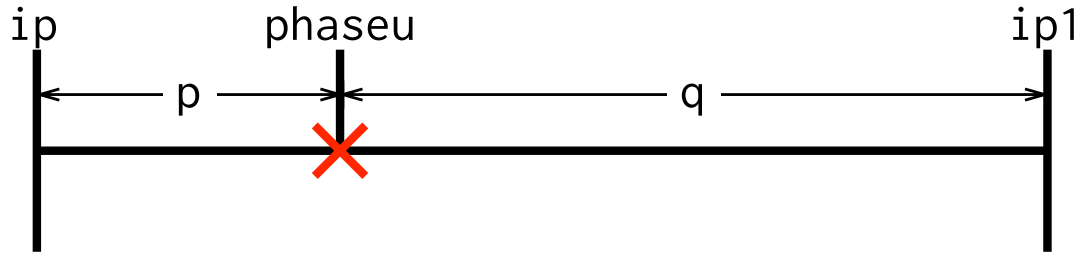


Figure 2.21: `k_func` interpolation. The value of `phaseu` is interpolated between `ip` and `ip1` using `p` and `q`.

The result is that $0 \leq$ `phaseu` ¡ `np`. This real number value will fall between the two rows of `array` to be read from. The integers `ip` and `ip1` are assigned the lower and upper row index respectively. `p` is assigned the difference between `ip` and `phaseu`, while `q` is assigned the difference between `phaseu` and `ip1`.

An interpolated value is calculated for each variable in `array` and assigned to the corresponding location in `u`.

If a single-line file was specified, the value of `phaseu` or `phasep` will have no effect on the values assigned to `u` or `p`.

Finally, before beginning the next iteration, the updated `u` is copied back to the current location in `New`.

2.12 Input/Output Devices

QUI provides several devices to output and analyse the results of a simulation. We will look at two of the simplest here; `record` and `ppmout`.

2.12.1 The `record` Device

The `record` device writes the values of selected dynamic variables to a file.

Once written, the file will begin with a header, specified by the `filehead` parameter, followed by a series of records; one for each run of the device. A record will contain values of the dynamic variables within the device's space, read in ascending order along the v,x,y and z axes in turn. Values are delimited in the file by separators, which mark a change in coordinate value for each of the axes.

The device takes 10 parameters, which are detailed in Table 2.9.

Type	Name	Default	Description
`int`	`append`	`1`	Flag indicating if the file should be appended to (`1`) or overwritten (`0`).
`str`	`file`	[Mandatory]	Relative path to the file that is to be written. Mandatory parameter.
`str`	`filehead`	`""`	Any content to be written once at the top of the file. By default, no file head is specified.
`str`	`recordhead`	`""`	Any content to be written for each record. By default, no record head is specified.
`str`	`vsep`	`" "`	String used to separate values of dynamic variables. Defaults to a space.
`str`	`xsep`	`", "`	String used between values from different x coordinates. Defaults to a comma followed by a space.
`str`	`ysep`	`"; "`	String used between values from different y coordinates. Defaults to a semicolon followed by a space.
`str`	`zsep`	`"n"`	String used between values from different z coordinates. Defaults to a newline character.
`str`	`format`	`"%lg"`	C format specifier to be used when printing values.
`str`	`recordsep`	`"n"`	String used between values from different records. Defaults to a newline character.

Table 2.9: Parameters of the `record` device.

The `file` parameter is mandatory; not providing a value will cause QUI to report an error and quit. All other parameters are optional.

2.12.2 The `ppmout` Device

The `ppmout` device produces Portable Pixel Map (*.ppm*) images derived from values within its space and writes them as a sequence of files on disk.

The colour of each pixel in the image is determined by the values of three dynamic variables, which are scaled to the range `0`–`255` and used as RGB colour values. For each colour channel, the dynamic variable is selected by its layer number and minimum and maximum value are set. The colour value for the channel is calculated as $255(value - min)/(max - min)$ where *value* is that of the dynamic variable. The same dynamic variables may be used for more than one colour channel. Minimum and maximum values must not be equal.

Each image produced by the `ppmout` device will be numbered in sequence. To enable this to happen for an indefinite number of files, the device's `file` parameter allows us to specify a mask; a template for each image's file name into which will be inserted a unique sequence

number. For example, in order to store the image files in a folder called *images* and label them *mySimulation-0000.ppm*, *mySimulation-0001.ppm*, *mySimulation-0002.ppm*, etc., the file `file` parameter should be assigned as `file=images/mySimulation%04d.ppm,-1`.

The `file` parameter takes a string that must include a C integer format specifier, `%d`. A unique sequence number will be inserted into the file name in place of the format specifier. In this case, we have used `%04d` to pad the number to four digits.

The number after the comma is the starting value for the sequence. In practice, a value of `-1` will produce *mySimulation-0000.ppm* as the first file, so use a value one less than your intended starting point. If files matching the mask already exist, the starting point will be set to the continue the sequence at the first opportunity. For example, if the user already has files numbered 1–5 and then 7–25, `ppmout` will start at 6 and overwrite any existing files thereafter.

With the exception of `append`, all of the parameters listed below must be given a value:

Type	Name	Description
`int`	`append`	Flag indicating if the file should be appended to (`1`) or overwritten (`0`).
`str`	`file`	Sequence mask of the image files to be written.
`int`	`r`	Dynamic variable number to be mapped to the red colour channel.
`int`	`r0`	Lower bound of dynamic variable `r`, equivalent to `0` in the red colour channel.
`int`	`r1`	Upper bound of dynamic variable `r`, equivalent to `255` in the red colour channel.
`int`	`g`	Dynamic variable number to be mapped to the green colour channel.
`int`	`g0`	Lower bound of dynamic variable `g`, equivalent to `0` in the green colour channel.
`int`	`g1`	Upper bound of dynamic variable `g`, equivalent to `255` in the green colour channel.
`int`	`b`	Dynamic variable number to be mapped to the blue colour channel.
`int`	`b0`	Lower bound of dynamic variable `b`, equivalent to `0` in the blue colour channel.
`int`	`b1`	Upper bound of dynamic variable `b`, equivalent to `255` in the blue colour channel.

Table 2.10: Parameters of the `ppmout` device.

2.12.3 The `dump` and `load` Devices

The `dump` device allows the user to save a snapshot of the state of the simulation medium to file in binary form. The `load` device allows the user to 'paste' the contents of such a file onto the simulation medium.

Both devices accept a string parameter called `file`, for the relative path to the file to be written to or read from. `dump` accepts an additional integer parameter, `append` that determines if the file will be appended (`1`) to or overwritten (`0`). When run, `dump` iterates over its space, writing a `double` value for each dynamic variable. Conversely, `load`, iterates over the same space, reading `double` values from file to dynamic variables.

2.13 Extending QUI

The device and RHS module APIs described in this chapter provide a well-defined means of extending QUI's functionality. Adding RHS modules from stand-alone codes is a relatively straightforward process [M^{c}Farlane, 2007].

Adding devices requires slightly more care to ensure that results are not adversely affected and that users see predictable behaviour from the parameter values specified. In particular, users will expect that condition and space parameters of the device are respected. It is reasonable for a device to assume that the timestep `t` will increase monotonically and that constants, such as the dimensions of the mesh, will not vary. Devices should also be safe to assume that the data in their parameter structures is well encapsulated. Since these properties cannot be enforced by the C language, QUI relies upon developers to code their devices responsibly.

The 'laws of devices' are listed more succinctly below. Each of these laws is broken at some point in the QUI code, but only with sufficient knowledge of the underlying system. Novice device developers are encouraged to adhere to the laws wherever possible.

1. A device should only alter layers of `New` in its space, at points in its space.
2. Aside from the permitted regions of `New`, a device should only alter the values of local variables or those in its parameter structure.
3. From any point in its space, a device should not reference points in `New` beyond $\{x \pm 1, y \pm 1, z \pm 1\}$.

2.14 Summary

In this chapter we examined the QUI software package in considerable detail. QUI provides a flexible, modular tool for the simulation of cardiac tissue. QUI's scripting language affords users with little computing experience the ability to compose simulations from modular devices and to parameterise those devices to their needs. QUI provides a broad selection of devices for computation and output, and a library of RHS modules allowing the user to choose their desired model of kinetics.

For users that require additional functionality, QUI's device and RHS APIs make it relatively simple to extend; most devices require the definition of a single `struct`, and three functions. Third-party developers are assisted in conforming to device conventions by a range of template and shortcut macros. We have also described a set of 'laws' that QUI devices should obey to ensure consistent, predictable behaviour.

The description of QUI provided by this chapter is the first known documentation of the code since its development around twenty years ago. This chapter forms the basis of our understanding of QUI, on which we can now build.

CHAPTER

THREE

BEATBOX: PARALLELISATION OF QUI USING MPI

3.1 Introduction

As discussed in Section 1.10, the complexity of cardiac simulations is steadily increasing. A single run of a typical simulation may take months on a high-specification desktop computer. When many runs are required, or when experimenting with different parameters, this becomes impractical. In order to achieve useful results in a reasonable amount of time, researchers are increasingly reliant on parallel computers.

The dominant form of parallel machine today is the distributed memory cluster. While a number of codes are now adopting hybrid shared/distributed memory models, this will not be developed at this stage. QUI does not presently have especially large memory requirements, and most of the memory is given over to `New`, which is the domain to be discretised. Given current RAM capacities of nodes in state of the art machines such as HECToR (Phase 2a: 2GB per node, Phase 2b: 1.33GB per node) and the rate of increase in cores per processor, it is likely to be some time before the reduction in RAM per node necessitates the use of mixed-mode programming. Shared memory programming usually requires more involved parallel design to avoid race conditions and deadlocks. To facilitate an extensible parallel architecture, in which third-party devices can benefit from parallelism implicitly, it is necessary to centralise parallelisation as much as possible. This is far more easily achieved using a distributed memory architecture.

3.1.1 Design Aims

The core aim of parallelisation is to greatly reduce the runtimes of large simulations by allowing users to harness the power of High Performance Computing (HPC) facilities. The code's ability to scale — i.e. its efficiency in using parallel processes — will be the main measure of its success. To achieve this, the parallelisation will have to resolve the conflicting goals of load balance and minimising communication.

As HPC is a specialist field in own right, it would demand a great deal of Beatbox users to learn and understand the details of its parallel implementation in order to use it. For this reason, Beatbox's parallel implementation should remain opaque to the user. User scripts should contain no data specific to their use in parallel, and the same script should be able to run safely and efficiently in both sequential and parallel modes. Such safety should not come at the expense of QUI's modular structure.

To a lesser extent, third-party developers should also be sheltered from the complexity of the parallel implementation. Third-party RHS modules and the majority of third-party devices should be able to benefit from parallelism implicitly, or with only minor modification to the code. Clear and concise guidelines should inform the user of any need to modify the code, and detail how to proceed.

To ease the burden of ongoing maintenance, the process of parallelisation should make minimal changes to the code, with as much code as possible shared between implementations. On systems where the parallel library is unavailable, the user should be able to compile a sequential-only executable.

3.1.2 MPI — The Message Passing Interface

MPI — The Message Passing Interface [Message Passing Interface Forum, 2003] is a FORTRAN and C/C++ library, providing tools for running programs on distributed memory machines. MPI provides an abstraction layer between the executable and the hardware implementation of the parallel machine.

Under MPI, an executable runs in one or more processes, where each process has its own, ring-fenced area of memory in which to operate. The MPI runtime environment declares the total number of processes, $size \in \mathbb{N},\ size \geq 1$, and assigns an identity, $rank \in \mathbb{N};\ 0 \leq rank < size$, to each process. Within an MPI program, processes may call MPI library functions to read or write files on disk, or to communicate with other processes. Functions in the MPI library can be either independent or collective. An independent function behaves in the same way as a normal C function. A collective function — such as one involving communication — requires the involvement of other processes, and will not return until the operation has completed on all of the processes involved. It is the blocking nature of collective operations that can cause delays in parallel programs, as processes wait for their slowest neighbour to complete its part of the task.

The processes involved in a collective operation are defined by a communicator, supplied as an argument to the function. All processes in an MPI program are members of the `MPI_COMM_WORLD` communicator. Additional communicators can be defined at runtime. This allows subsets of processes to take part in collective operations, which can improve efficiency and affords additional flexibility. Within a communicator, processes are numbered from 0, meaning that processes may have different ranks in each of the communicators of which it is a member.

3.2 Domain Decomposition

Beatbox benefits from parallel processes by employing *domain decomposition*, in which each processor handles a portion of the mesh. The mesh is divided between the processes by overlaying a coarse-grained *supergrid*. Each region of the supergrid defines a *subdomain* of the medium that will be assigned to one process.

The dimensions of the supergrid are computed to fit the dimensions of the simulation medium and the number of available processes, with the aim to improve performance by striking a balance between making use of all available processes, balancing load and minimising the communication between processes. A discussion of how supergrid dimensions are computed is given in Section 3.2.4.

Using the process described in Section 3.2.4, one or more processes is assigned to each axis of the supergrid, where nx, ny, nz are the number of processes assigned to the x, y and z axes respectively. The resulting *subdomains* can be identified by a triple of *superindices*, $(ix, iy, iz) \in \mathbb{N}; 0 \leq ix < nx, 0 \leq iy < ny, 0 \leq iz < nz$. The above variables are made available globally via the `state` module (*state.h*), prefixed with `mpi_`.

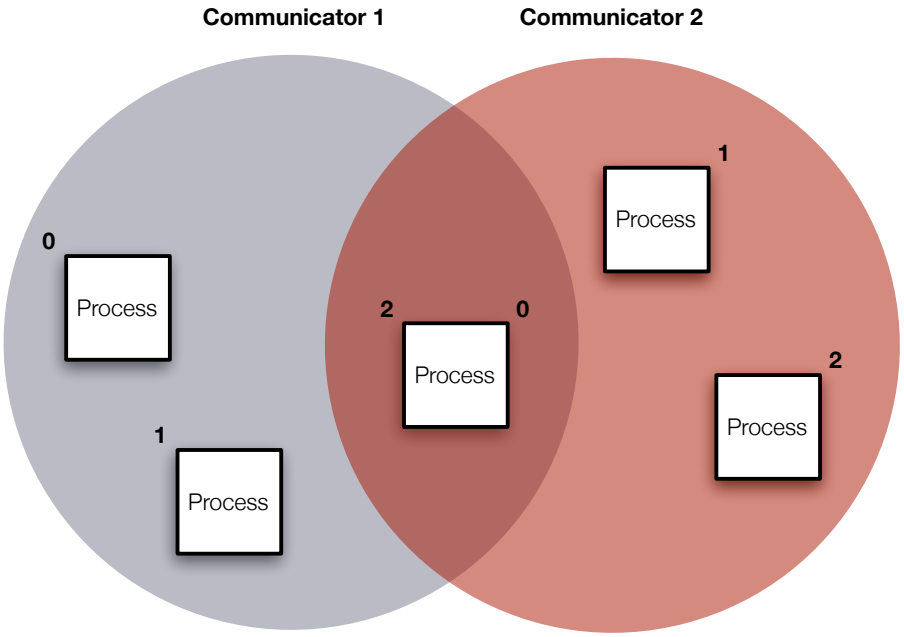


Figure 3.1: Processes, communicators and ranks. Processes may exist in more than one communicator. A process is assigned a rank in every communicator of which it is a member.

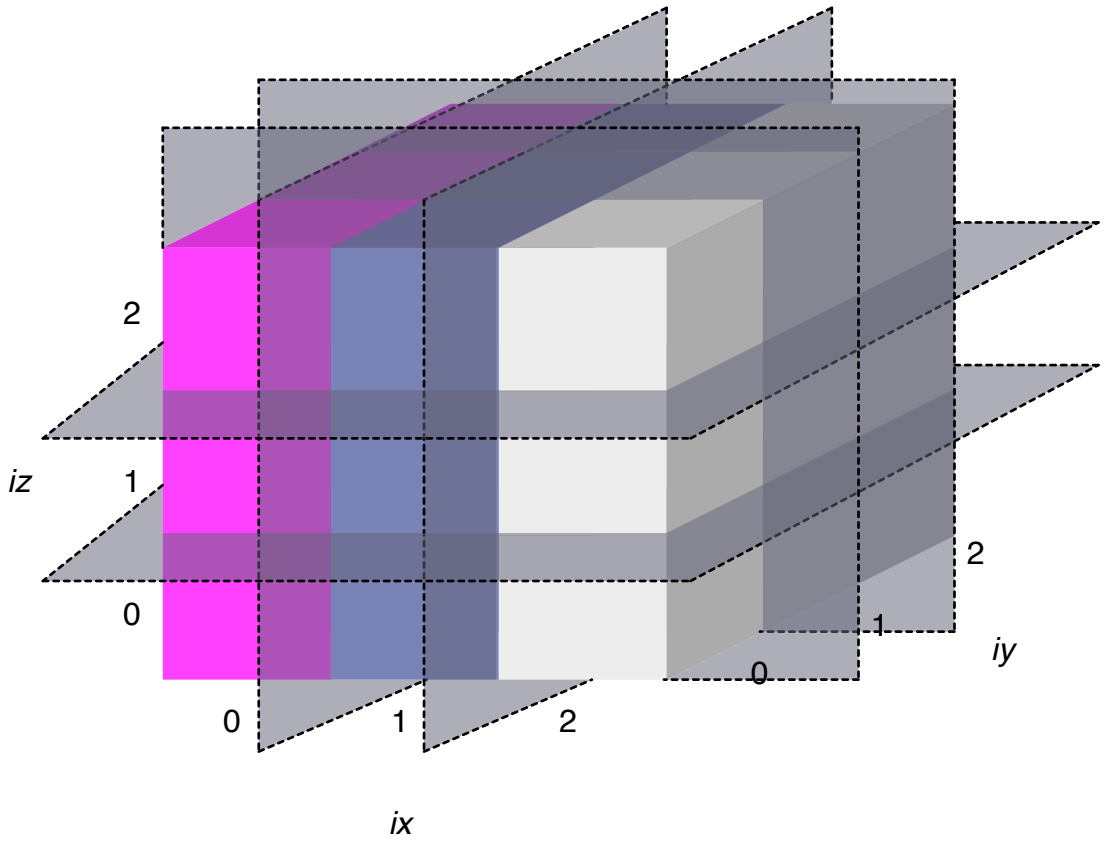


Figure 3.2: Domain Decomposition Supergrid with Superindices

3.2.1 Assigning Processes on the Supergrid

The DETERMINESUPERINDICES procedure, below, defines the relationship between a process's *rank* and the superindices of its subdomain:

DETERMINESUPERINDICES(*rank*)
1 $ix \leftarrow rank\,/(ny \times nz)$
  ▷ rt is the remainder
2 $rt \leftarrow rank \mod (ny \times nz)$
3 $iy \leftarrow rt\,/\,nz$
4 $iz \leftarrow rt \mod nz$

DETERMINESUPERINDICES is implemented as part of `state()` (*state.c*) and is shown in Listing C.1.

GETPROCESSRANK does the opposite of DETERMINESUPERINDICES, taking superindices and returning the process rank:

GETPROCESSRANK(ix, iy, iz)
1 $rank \leftarrow (ix \times ny \times nz) + (iy \times nz) + ix$
2 **return** *rank*

GETPROCESSRANK is defined in *state.c* and made accessible via *state.h*:

```
int getProcessRank(int ix,int iy,int iz){
    return (ix*mpi_ny*mpi_nz)+(iy*mpi_nz)+iz;
}
```

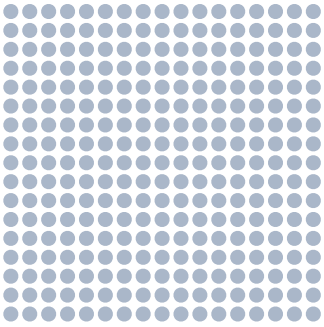
3.2.2 Defining Subdomains

Active Axes and Internal Points

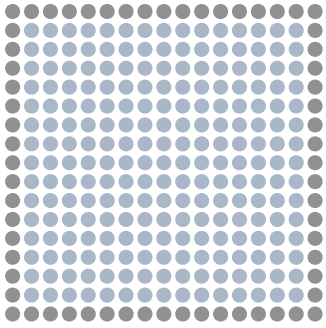
The decomposition of a domain will depend on the size and shape of the domain to be decomposed. The dimensionality of the mesh must be considered to ensure that decomposition only occurs along axes that are in use. Crucial to this process is the concept of an active axis. An axis is considered to be active if its maximum (`xmax`, `ymax` or `zmax`) is greater than or equal to 3.

The reason that three points is significant with respect to the length of an axis is due to the way points at the extremes of the axis are used. In QUI, points at the extreme of the axis would be updated by a boundary conditions device, such as `neum3d` or `dirich3d`. The remaining points, i.e. those contained by the default space, would be operated on by the `euler` and `diff3d` devices. Because the stencil of the diffusion device will cause it to reference immediate neighbours, it is important that the diffusion device does not operate on points at the extremes of the medium, lest it refer to non-existent points beyond the mesh's edges. For this reason, we make the distinction between internal and boundary points on an axis. Boundary points have coordinates of 0 and $imax - 1$. Internal points range from $1 \leq i \leq imax - 2$. We identify an axis as active if there are three or more points as this allows for at least one internal point, with a boundary point at either end. Figure 3.3a shows points defined by the user and Figure 3.3b shows internal and boundary points.

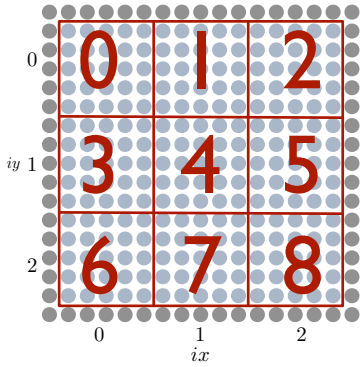
To avoid ambiguity regarding boundary allocation, the MPI implementation adds a condition to `state()` that if the user specifies exactly 2 points along any axis, an error will be raised and Beatbox will quit.



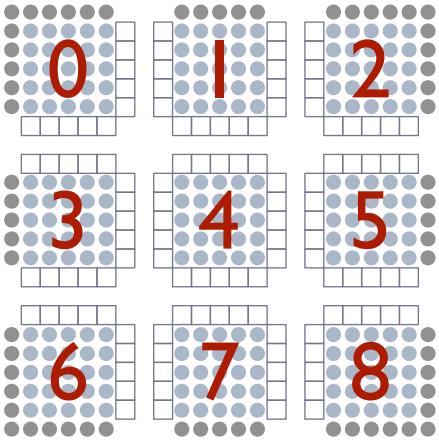
(a) The mesh.



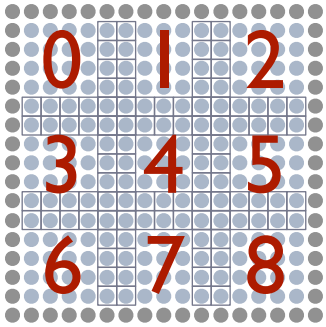
(b) Mesh with boundary points identified.



(c) Mesh with overlaid supergrid.



(d) Subdomains with halos.



(e) Subdomains united.

Figure 3.3: Domain decomposition in two dimensions

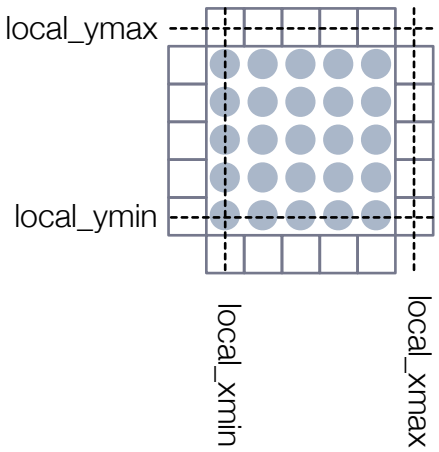


Figure 3.4: Limits of the local subdomain.

Subdomain Bounds

The MPI implementation aims, wherever possible, to maintain the illusion for devices and users that the program is running sequentially. Although local subdomains contain only a subset of the points in the simulation medium, the values held at those points are accessed by the same coordinates as would be used in the sequential version. To achieve this, each process must know the area of the mesh represented by its local subdomain.

To define the area on which each process will operate, the MPI implementation adds local minima and maxima for each process. These determine the range of coordinates of *internal* points for that subdomain. For example, internal points in the local subdomain are those with coordinates $local\_imin \leq i < local\_imax$, as illustrated in Figure 3.4.

The values of local minima and maxima are arrived at by dividing *internal* points of the medium between processes, as illustrated in Figure 3.3c. Pseudocode for obtaining subdomain dimensions on the x axis is given in GetSubdomainDimensions. Code for the process is given in Listing C.2.

GetSubdomainDimensions()

▷ Compute approximate slice size
1 $x\_slice \leftarrow (\text{XMAX} - (\text{ONE} \times 2))/\text{NX})$
▷ Remaining points will be shared amongst the first processes on the axis
2 $x\_remainder \leftarrow (\text{XMAX} - (\text{ONE} \times 2))mod(\text{NX})$
3 **if** $ix < x\_remainder$
4 **then**
5 $local\_xmin \leftarrow ONE + (ix \times x\_slice) + ix$
6 **else**
7 $local\_xmin \leftarrow ONE + (ix \times x\_slice) + x\_remainder$
8 **if** $ix < x\_remainder$
9 **then**
10 $local\_xmax \leftarrow ONE + ((ix + 1) \times x\_slice) + (ix + 1)$
11 **else**
12 $local\_xmax \leftarrow ONE + ((ix + 1) \times x\_slice) + x\_remainder$
▷ Repeat for y and z axes...

After decomposition, the union of internal points from all subdomains will be equivalent to the default space. As shown in Figure 3.3d, around each subdomain is a single layer of points, referred to as a *halo*, which performs two roles. Where halo points lie at the ends of an axis, they will act as the boundary points of the simulation medium. Halo points on internal edges of a subdomain are used to hold local copies of points from neighbouring processes. The remainder of the halo points will be used to hold copies of values from neighbouring subdomains, as described in Section 3.3. Placing points outside the default space in the halos in this way allows for more consistent behaviour when adapting devices to run on decomposed subdomains. This is discussed in more detail in Section 3.2.3.

Accessing `New` in Parallel

Accessing data in the simulation medium requires mapping the coordinates of the point being accessed to the corresponding array index in `New`. As in QUI, this mapping is performed by the `ind()` function macro (*state.h*). The MPI implementation maintains the coordinate space of the simulation medium by providing an alternative version of `ind()`, conditionally compiled, to take account of the subdomain's position in the mesh. Pseudocode for the parallelised `ind()` macro is given in the ParallelInd algorithm, below.

ParallelInd(x, y, z, v)

1 **return** $((x + \text{ONE}) - local\_xmin) \times \text{VMAX\_ZMAX\_YMAX} +$
$((y + \text{TWO}) - local\_ymin \times \text{VMAX\_ZMAX} +$
$((z + \text{TRI}) - local\_zmin) \times \text{VMAX} + v$

Figure 3.3e illustrates how the medium can be reconstructed from its subdomains. Overlapping points, where dots are contained in boxes, will have the same coordinate values in both subdomains, although every point will be *internal* to only one process. As described in the Section 3.3, halo points on the internal edges of subdomains (boxes) will hold a copy of the values from their corresponding internal point (blue dot).

3.2.3 Constraining Device Spaces

As described above, every device defined in the user script will be created in every process. To prevent the device accessing points not available locally and to avoid the same work from being executed on every process, the effects of each device must be constrained to its local subdomain. Commonly, a device iterating over the whole medium would do so using the fields of its space structure as loop bounds:

```
for(x=s.x0;x<=s.x1;x++) {
    for(y=s.y0;y<=s.y1;y++) {
        for(z=s.z0;z<=s.z1;z++) {
            // DO THIS AT EVERY POINT
        }
    }
}
```

QUI already provides a mechanism for limiting the scope of devices, in the form of `Space` structures. The MPI implementation of Beatbox adapts `Space` structures such that loops like these will iterate over only points in the local subdomain. Discussion of the MPI implementation requires some additional terminology. We refer to the space defined in the script as the *global space* and this is identical to the device's space when running sequentially. The intersection of the global space and local subdomain is referred to as the *local space*.

There are several use cases, described below, that require a device to know both its local space and the global space of which it is part. For this reason, the MPI implementation adds fields prefixed with `global_` to the `Space` structures definition to hold this data. The fields of the parallelised `Space` structure are listed in Table 3.1 The additional `runHere` field is introduced to indicate if the device should be run in the local subdomain. The `nowhere` parameter is added to make specification of a device as *nowhere* more explicit.

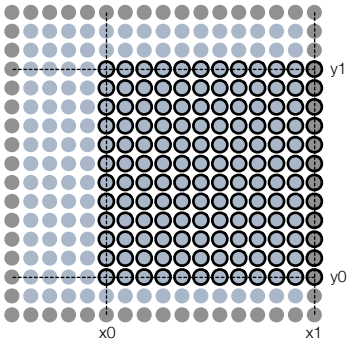
The `nowhere` Parameter

In QUI, all devices have a `Space` structure to describe the area of the mesh on which they will operate. Some devices, however, do not operate on the mesh directly, but are instead required to perform some kind of global operation. For example, when `k_func` devices are used to update timing variables, they do not operate on the mesh, although they have a space. By convention in QUI, users could give the device space parameters using the `nowhere` macro (*std.qui*), which expands to `x0=0 x1=0 y0=0 y1=0 z0=0 z1=0`.

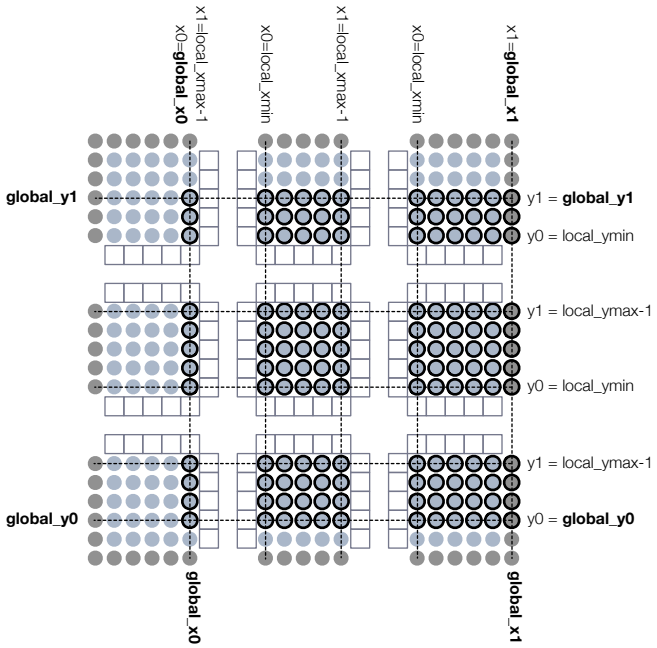
In the MPI implementation, the space declared by the `nowhere` macro intersects with only one process, thus preventing its device from running on any other processes. This is clearly not what the user should expect. It would be possible to interpret the space defined by the `nowhere` macro as an indication that the device should not operate on the mesh, however this causes problems in zero-dimensional simulations, where `x0=0 x1=0 y0=0 y1=0 z0=0 z1=0` is the default space.

To solve these problems, and to remove the ambiguity surrounding devices without an explicit space on the mesh, the MPI implementation introduces `nowhere` as a space parameter, available to all devices. For consistency, `Nowhere` is added to the `Space` structure (*device.h*) for both sequential and parallel versions. The user can declare a device to be 'nowhere' by adding `nowhere=1` to the device call. There is, however, the possibility of further ambiguity if the user specifies both `nowhere=1` and other space parameters. As discussed below, this is handled in `accept_space()` (*qpp.c*).

The effect of the `nowhere` parameter on the running of devices is discussed below. The `nowhere` parameter is also particularly important in the parallel implementation of the `k_func`



(a) Sequential space



(b) Parallel spaces

Figure 3.5: Relationship between spaces in sequential and parallel implementations. 3.5a shows space parameters as they relate to the global simulation medium. Points included in the space are outlined in black. 3.5b shows global (bold) and local space parameters as they relate to the local limits of internity (i.e. `local_xmin`, `local_xmax`, etc.). Local spaces are defined as the intersection of the local subdomain and the global space; including absolute, but not internal boundaries.

Type	Name	Description
Sequential and MPI		
`int`	`x0`	Minimum x value accessible by the local device instance.
`int`	`x1`	Maximum x value accessible by the local device instance.
`int`	`y0`	Minimum y value accessible by the local device instance.
`int`	`y1`	Maximum y value accessible by the local device instance.
`int`	`z0`	Minimum z value accessible by the local device instance.
`int`	`z1`	Maximum z value accessible by the local device instance.
`int`	`v0`	Minimum v value accessible by the local device instance.
`int`	`v1`	Maximum v value accessible by the local device instance.
`int`	`nowhere`	Indicates if the device is *nowhere*, i.e. doesn't operate on the simulation medium.
MPI Only		
`int`	`global_x0`	Minimum x value accessible by any device instance.
`int`	`global_x1`	Maximum x value accessible by any device instance.
`int`	`global_y0`	Minimum y value accessible by any device instance.
`int`	`global_y1`	Maximum y value accessible by any device instance.
`int`	`global_z0`	Minimum z value accessible by any device instance.
`int`	`global_z1`	Maximum z value accessible by any device instance.
`int`	`runhere`	Indicates if the local device instance should run.

Table 3.1: Fields of the parallelised `Space` structure.

device, discussed in Section 3.6.

Accepting Device Spaces

As described in Section 2.4.2, the `accept_space()` function (*qpp.c*) is responsible for parsing the user-specified space parameters from the script and setting values in the `Space` structure accordingly. The MPI implementation adapts this process for decomposed domains.

The first property read from the script by `accept_space()` is the new `nowhere` parameter. If `nowhere` is set to `1` (TRUE), all space parameters are assigned the minimum value, corresponding to the first corner of either the simulation medium when running sequentially, or the local subdomain when running under MPI. If `nowhere` is set to `1`, the existence of any space parameters (`x0`,`x1`,`y0`,`y1`,`z0`,`z1`) in the script will raise an error.

When `nowhere` is `0` (FALSE), the MPI implementation accepts space parameters from the script in exactly the same way as the sequential version. The `x0`, `x1`, etc., fields of the `Space` structure are assigned values from the script, with the default values if none are provided. An error is raised if the given value falls outside of the simulation medium (i.e. $\leq$ `0` or $>$ `max-1`), or if upper and lower limits are transposed (e.g. `x0` $>$ `x1`).

The MPI version of the function then assigns the accepted values, which define the *global space*, to the `global_` fields of the `Space` structure. A series of checks then adjust the local `Space` parameters (`x0`,`x1`, etc.) for the local subdomain, as illustrated in Figures 3.6 and 3.5b. If a device's space falls completely outside the local subdomain it should not be run in this process, so `runHere` is set to `0` (FALSE). Finally, if the device's space intersects the local subdomain, the space parameters are adjusted to give the intersection of the global space and local subdomain.

Because the absolute boundary points fall into subdomains' halos, a special case occurs when

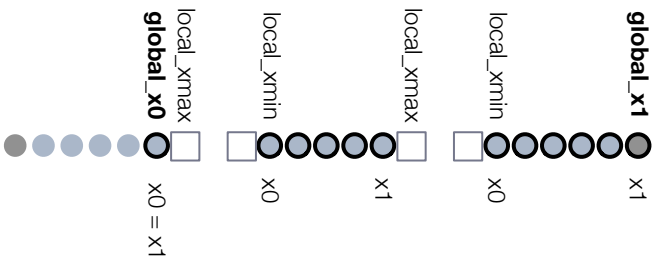


Figure 3.6: Relationship between global and local spaces. A single row on the x axis is shown. Global space parameters for the x axis are shown in bold. Points included in the space are outlined in black. Note that halo points are not included in the space, except where they form part of an absolute boundary (i.e. `x = local_xmax-1`).

the device's global space contains absolute boundary points, i.e those with a coordinate $i = 0$ or $i = imax - 1$. Although boundary points strictly lie outside the local subdomain, unlike internal halo points they each exist in only one process. If required, those processes should include them in their devices' local spaces. This is illustrated in Figures 3.5 and 3.6. In the rare case when a device's global space *only* occupies a boundary, both of its space coordinates on the corresponding axis will appear to fall outside of *every* local subdomain, as shown in Figure 3.7. In this case, device instances on processes that 'own' the boundary points in question should include them in their local space, and be permitted to run by setting their `runHere` flag to `1`.

On processes where the device's space does not intersect with the local subdomain or its absolute boundaries, the device should be prevented from running. This will avoid any unintended effects caused by device code running inappropriately, and may yield a small performance improvement. The `RUN_HEAD` and `RUN_TAIL` macros (*device.h*) are altered in the MPI implementation to test `runHere` prior to running the function body:

```
#if MPI
#define RUN_HEAD(name)   \
PROC(run_##name) {    \
    if(sync){haloSwap();}  \
    if(s.runHere || alwaysRun){ \
        STR *S = (STR *) par;
```

Listing 3.1: *device.h* — MPI version of the device `RUN_HEAD` macro.

```
#define RUN_TAIL(name)  \
    } /* if s.runHere */ \
    return 1;    \
}
```

Listing 3.2: *device.h* — MPI version of the device `RUN_TAIL` macro.

The test is made inside the Run function, after the `haloSwap()` call, to allow collective communication to take place using all active processes. This is described in greater detail in Section 3.3.1.

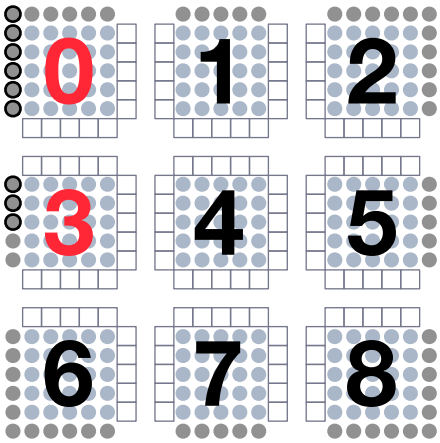


Figure 3.7: Example of a space that exists only on an absolute boundary. As 'owners' of the absolute boundary points, processes 0 and 3 (shown in red) will have their `runHere` flags set to true, so that they can operate on the space.

3.2.4 Computing Supergrid Dimensions

The `decompose()` (*decomp.c*) function computes the dimensions of the supergrid, given the dimensions of the mesh, and the number of available processes. `decomp()` operates in a brute-force fashion, considering each of the possible configurations of supergrid dimensions, with some restrictions:

- Exactly one process must be assigned to any inactive axis.
- The number of processes assigned to one axis must not exceed the number of points along that axis.
- The total number of processes must not be greater than `mpi_size`, but may be less.

The basis of their comparison is a metric computed by `decomp_getMetric()`:

GetMetric(*decomp*)

▷ *decomp* is a structure containing the *nx*, *ny* and *nz* for the candidate decomposition.
1 **return** (WorstVolume(*decomp*) × VOLUME_ADJUST+
WorstSurfaceArea(*decomp*) × SURFACE_AREA_ADJUST)

The majority of simulations will vary little in computational complexity between points of the mesh, so it can be assumed that the number of points in a subdomain is representative of its computational load. Similarly, the number of points on a subdomain's surface area is likely to correspond with the volume of communication handled by its process.

Each decomposition is scored by `decomp_getMetric()`, with lower scores being preferred. The values *VOLUME_ADJUST* and *SURFACE_AREA_ADJUST* shift the objectives of the decomposition process. A relatively high *VOLUME_ADJUST* will cause decompositions containing large subdomain volumes to have higher scores. This will bias the decision in favour of decompositions that spread load evenly between processes, with a greater likelihood that most or all available processes will be used, at the expense of communication. Conversely, a

relatively high *SURFACE_AREA_ADJUST* will increase the score of decompositions containing large surface areas. Consequently, the algorithm will favour more cube-like subdomains, so as to reduce communication, but this may require that some processes remain idle. Current settings of *VOLUME_ADJUST* = 1 and *SURFACE_AREA_ADJUST* = 100 are heuristic values that yield appropriate results for most domains.

WorstVolume and WorstSurfaceArea compute the volume and surface area, respectively, of the worst-case subdomain in the given decomposition. Both functions make use of WorstDimensions():

```
WorstDimensions(decomp)
1  if xmax mod decomp.nx = 0
2     then
3          xlen = (xmax/ decomp . nx)
4     else
5          xlen = (xmax/ decomp . nx) + 1
   ▷ Repeat for xlen and ylen ...
6  return {xlen, ylen, zlen}
```

```
WorstVolume(decomp)
1  {xlen, ylen, zlen} ← WorstDimensions(decomp)
2  return xlen × ylen × zlen
```

```
WorstVolume(decomp)
1  {xlen, ylen, zlen} ← WorstDimensions(decomp)
2  return 2 × (xlen × ylen + ylen × zlen + xlen × zlen)
```

The decomposition process may choose to leave some processes idle, in order to reduce communication. It is important that idle processes are effectively isolated, otherwise calls to collective MPI functions would result in deadlock, where active processes wait indefinitely on the involvement of idle processes. To separate active and idle processes, a new MPI communicator, `ALL_ACTIVE_PROCS`, is created, including only those processes that have been assigned a subdomain. The flag `I_AM_IDLE` is set to indicate an idle process, and is later tested in `main()` (*qui.c*) to prevent idle processes from creating or running devices. While the simulation is running, idle processes will wait at an `MPI_Barrier` call in the `Exit:` section of *beatbox.c*. When the active processes complete the simulation, they will meet the idle processes at this barrier before exiting the program together.

3.3 Inter-Process Communication

While the adaptation of `Space` structures is sufficient to parallelise many of QUI's devices for MPI, some devices, such as those that compute diffusion, rely upon the propagation of information from one point in the medium to another. This will necessarily involve devices referencing points from neighbouring subdomains. Since these points cannot be read directly from other processes, we must exchange them explicitly between processes and retain copies of the relevant neighbouring values locally.

Figure 3.8 shows the subdomains of neighbouring processes, separated to show the surfaces that will be commmunicated. In order for Process 9's diffusion device to read data from points to its left, it receives a copy of the (y, z, v) hyperplane (shown in red) from Process 0's right. In

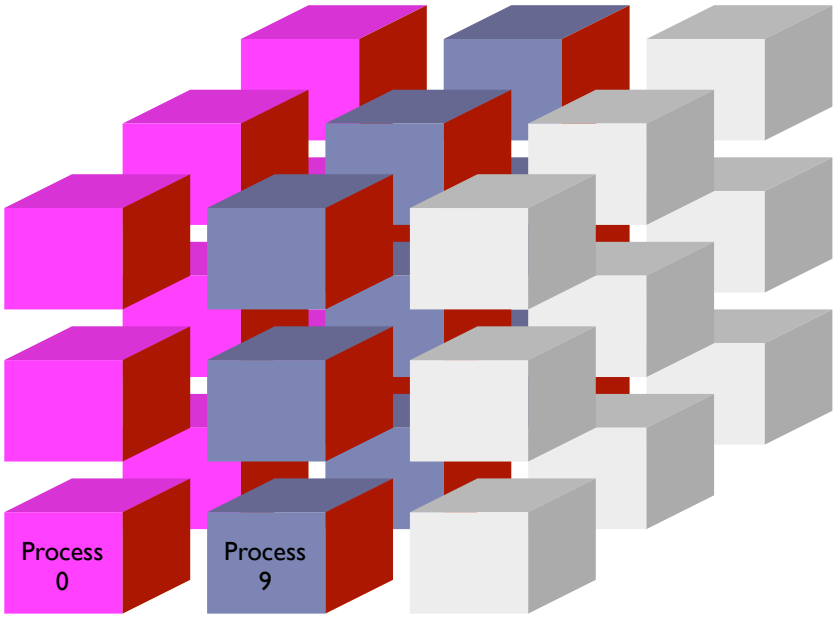


Figure 3.8: Hyperplanes to be Communicated Along the x Axis: Red surfaces will be exchanged between neighbouring processes.

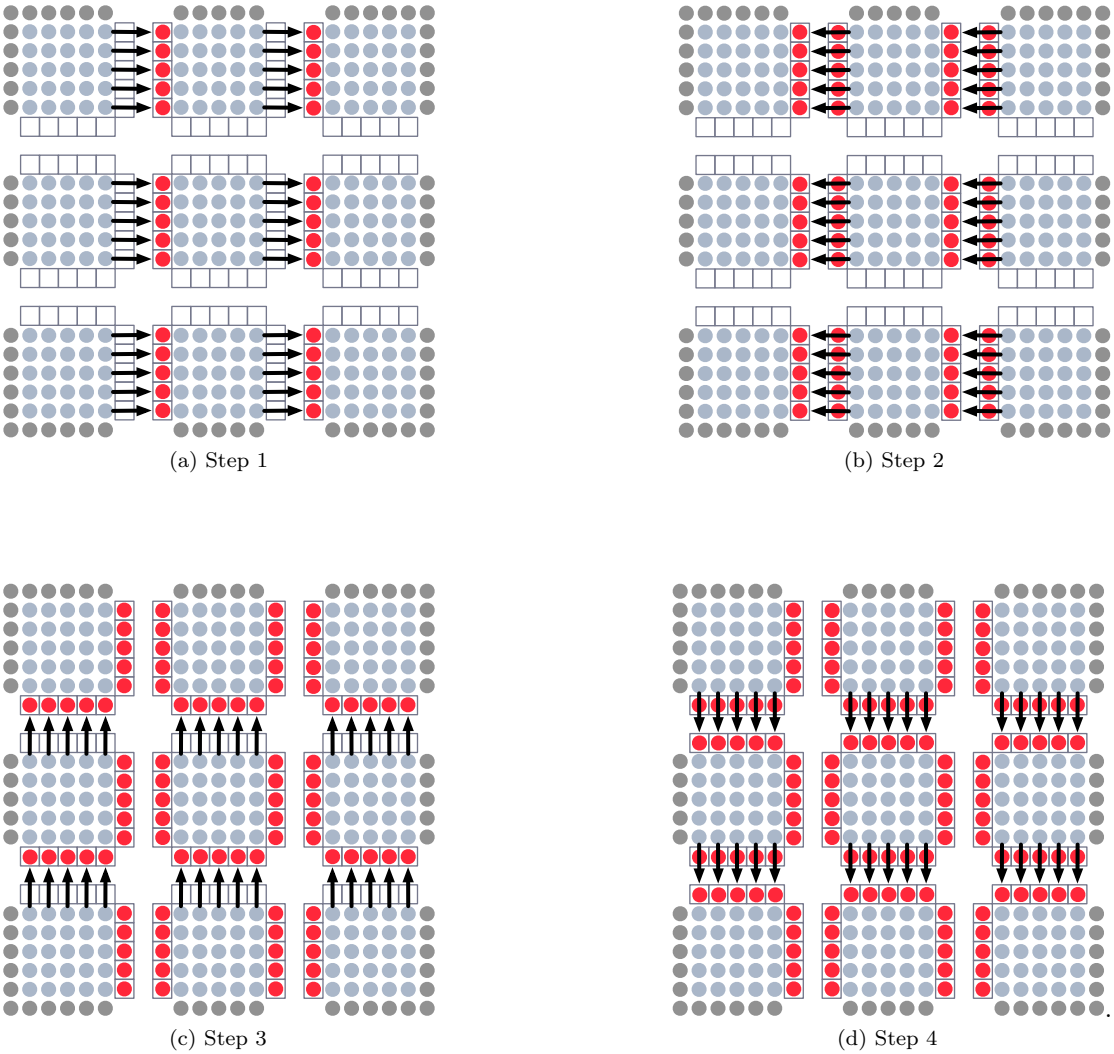
return, Process 0 will receive a copy of Process 9's leftmost (y, z, v) hyperplane. Pseudocode for this operation is shown below. To prevent the blocking `MPI_Sendrecv()` calls causing deadlock, calls are paired between processes with odd and even ranks, as described below:

HaloSwap

```
1  if (ix  mod 2) = 0
2     then
3          if ix < nx −1
4            then
5                 Send (y, z, v) hyperplane at local_xmax-1 to XP_NEIGHBOUR
6                 Receive into (y, z, v) hyperplane at local_xmax from XP_NEIGHBOUR
7     else
          ▷ No need to check if ix > 0, since ix is odd.
8         Receive into (y, z, v) hyperplane at local_xmin-1 from XN_NEIGHBOUR
9         Send (y, z, v) hyperplane at local_xmin to XN_NEIGHBOUR
10 if (ix  mod 2) = 1
11    then
12         if ix < nx −1
13           then
14                Send (y, z, v) hyperplane at local_xmax-1 to XP_NEIGHBOUR
15                Receive into (y, z, v) hyperplane at local_xmax from XP_NEIGHBOUR
16    else if ix > 0
17           then
18                Receive into (y, z, v) hyperplane at local_xmin-1 from XN_NEIGHBOUR
19                Send (y, z, v) hyperplane at local_xmin to XN_NEIGHBOUR
   ▷ Repeat for y and z axes...
```

Process 0	Process 1	Process 2	Process 3
→	⊐	→	⊐
⊏	←	⊏	←
	→	⊐	
	⊏	←	

Table 3.2: Pairing of `MPI_Send()` and `MPI_Recv()` calls



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 3.9: Two-dimensional view of the Beatbox halo exchange.

3.3.1 `haloSwap()`

The communication process described by HaloSwap is implemented in the `haloSwap()` function of *device.c*, the full code for which is shown in Listing C.3. In practice, the separate send and receive actions shown above are combined in the `MPI_Sendrecv()` function. This prevents the need to establish separate connections to send and receive with the same neighbouring process.

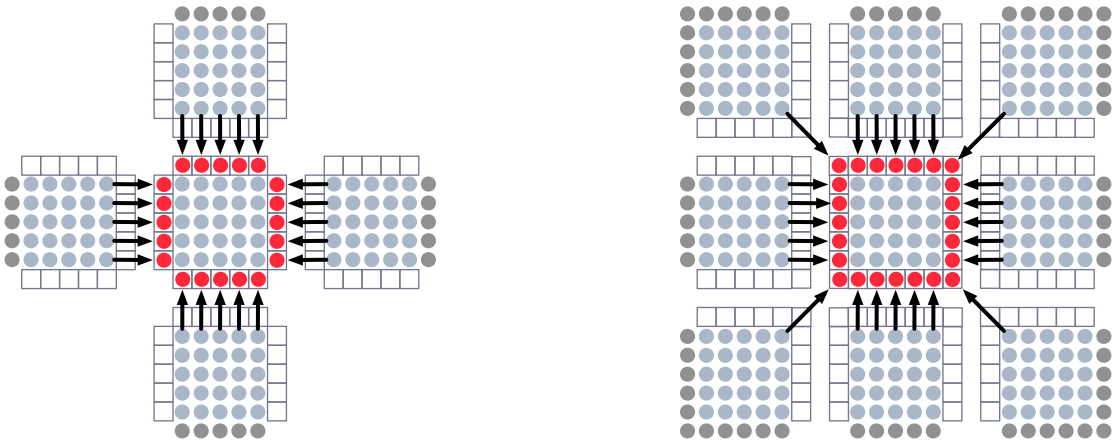
Defining Hyperplanes for Communication

Each subdomain will have up to six surfaces that require halo exchange with surrounding processes. For each surface, two three-dimensional hyperplanes of `New` are defined — the surface of the subdomain to be sent, and the corresponding halo into which received data will be written. It is possible to facilitate the exchange of data between processes by buffering, then 'unpacking', the received data into its destination; early versions of Beatbox employed just such a method.

The MPI library, however, offers an efficient mechanism by which buffering can be avoided. To the developer, data appears to be directly sent from, and received into the required regions of an array. To do this, MPI requires that the regions of the array being accessed are defined as a derived datatype, using subarrays. A subarray allows a one-dimensional array to be viewed as containing n-dimensional data, and to place constraints on that array, such that sparsely arranged data within the array can be accessed as if contiguous in memory. The concept of subarrays fits Beatbox's use of `New` well. Subarrays for halo-swapping are defined in the `decomp_defineHaloTypes()` function of *decomp.c*, which is shown in Listing C.4. Once defined, `MPI_Sendrecv()` calls can access `New` using the derived datatype definition, allowing subarrays to be sent or received in a single call. Examples of derived datatypes in use can be seen in Listing C.3.

Magic Corners

As discussed in Section 4.11, some operations used with anatomically realistic tissue geometry must be able to reference neighbouring points diagonally. For points at the corners of the local subdomain, this will mean referencing points from diagonally neighbouring subdomains. A 2D subdomain will have 4 diagonal neighbours in addition to its 4 orthogonal neighbours, as illustrated in Figure 3.10. In 3D, 20 diagonal neighbours are added to the existing 6 orthogonal neighbours. Since there is significant overhead in establishing communication to other processes, synchronisation with these additional processes every timestep is to be avoided.



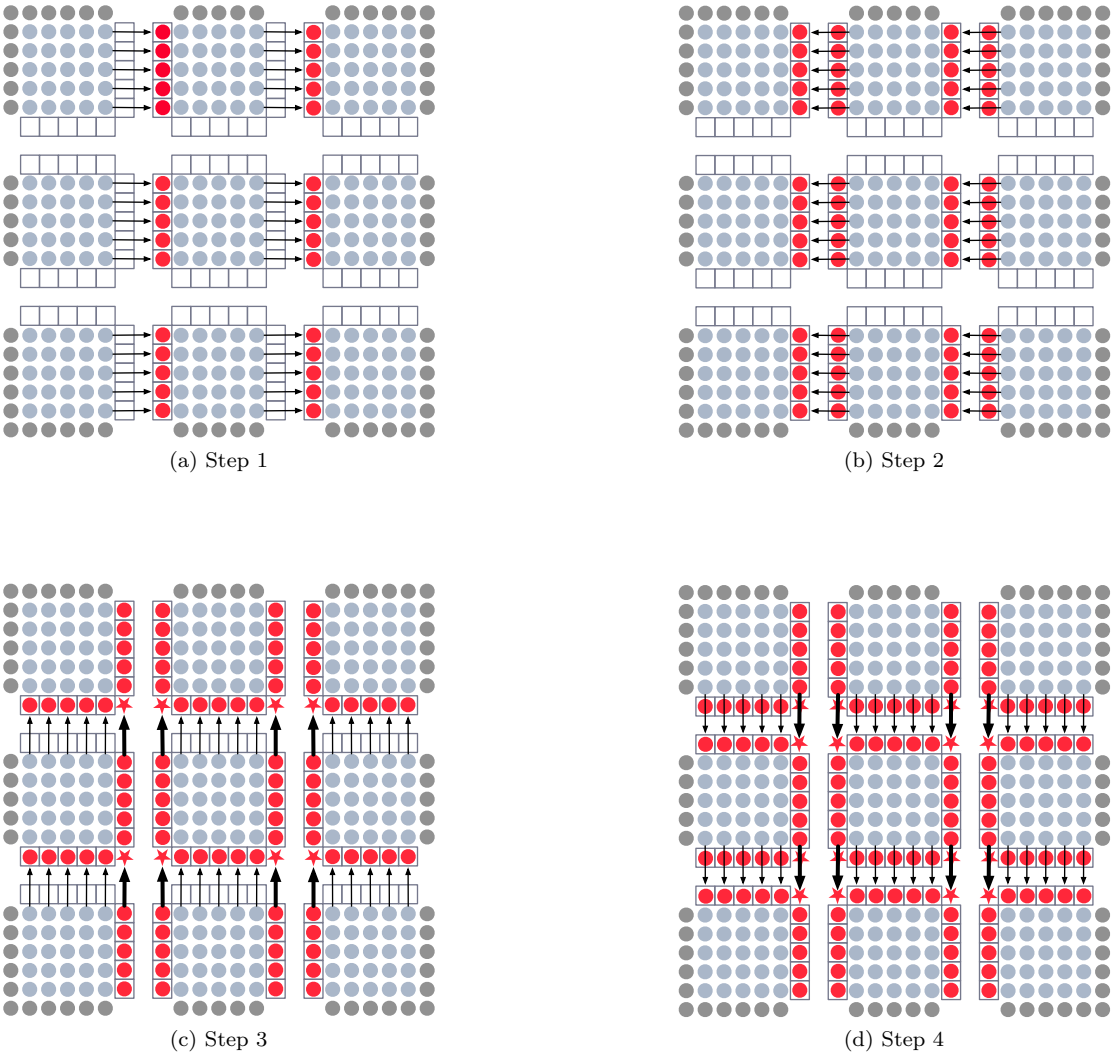
(a) Halo exchange involving only orthogonally neighbouring processes.

(b) Halo exchange involving orthogonally and diagonally neighbouring processes.

Figure 3.10: Two-dimensional views of traditional halo exchanges. Red dots indicate updated halo points.

Beatbox employs an algorithm designed to obviate the need to synchronise with diagonally neighbouring subdomains. Instead of communicating individual corner points between diagonally neighbouring processes, we shall exchange orthogonally the extended corresponding faces of the local domain hypercube, (x, y, z, v). First, we widen the (x, z, v) face of the local subdomain to include the halo points in the x direction. We then exchange these widened faces in the y direction. We then widen the (x, y, v) faces to include halo points in both the x and y directions. These faces are then exchanged in the z direction. As illustrated in Figure 3.11, exchanging widened faces of the local subdomains in this way, in the correct sequence (the process relies upon

exchanges taking place in x,y,z order), allows corner points to travel between processes indirectly, via the 6 orthogonal neighbours. The corner points updated by this method are referred to as 'magic corners'. Magic corners will always travel along axes in x,y,z order, meaning that, for a point to come from $\{x-1, y+1, z+1\}$, it will travel in the negative direction along the x axis, then positively along the y axis, then positively along the z axis. Its reciprocal point will travel positively along the x axis, negatively along the y axis, then negatively along the z axis to reach its destination. The path taken by two opposing corner points is illustrated in Figure 3.12



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 3.11: Two-dimensional view of the Beatbox halo exchange, with 'Magic Corners' shown as stars. New exchanges are shown with larger arrows.

Enabling Collective Communication

When called, `haloSwap()` starts a collective exchange of halos. It is necessary to call `haloSwap()` on every active process to ensure that all points that could be referenced from within a device's

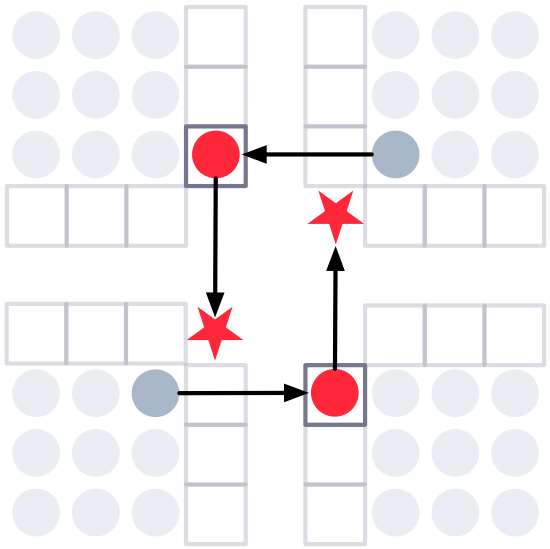


Figure 3.12: Close-up view of the path taken by two opposing corner points. Travel is along the x axis, then the y axis.

global space are updated. In order for this to work, `haloSwap()` must be called by every instance of a device, including those whose `runHere` flag is set to `0`. This is managed by inserting calls to `haloSwap()` inside the device's Run function using the `RUN_HEAD` macro (*device.h*). As shown in Listing 3.1, `haloSwap()` is called before `runHere` is tested. This ensures that every active process is involved in exchanging halos.

Preventing Unnecessary Communication

Exchanging halos is an expensive operation, due to both the time taken to send and receive data across the network, and the 'synchronisation time' taken as collective calls require processes wait for the slowest process to catch up. Only devices that reference points outside of their space — i.e. those with 'stencils' of more than one point — need call `haloSwap()`. Devices indicate their need for synchronisation by setting the `sync` field of their `Device` structure to `1`. As shown in Listing 3.1, the `RUN_HEAD` macro tests `sync`before calling `haloSwap()`.

3.3.2 Device-Specific Communication

In addition to global halo exchanges, some devices may require further inter-process communication, e.g. to share the result of a computation, or to perform a reduction operation across the device's global space. This section discusses the common functions and techniques available to devices and device developers who wish to implement communication within a device; specific applications are discussed in Section 3.4.

In order to perform collective communications within a device's Run function, only active instances of a device can be involved. This is achieved by defining an MPI communicator over processes on which the device's `runHere` flag is set to `1`. This functionality is provided in *(*device.c), via the `deviceCommunicator()` and `deviceCommunicatorWithFirstRank()` functions. Both functions create a new communicator containing all processes on which the device's `runHere` flag is set to `1`. For devices that require a 'root' process for one-to-many or many-to-one operations,

such as broadcast, scattering, gathering or reduction, `deviceCommunicatorWithFirstRank()` can also return the rank of the first process in the communicator. Code for both functions is given in Listing C.5.

3.4 Preparing Devices for Parallelism

The alterations described above will provide *implicit parallelism* to the majority of devices, i.e. those devices will benefit from Beatbox's parallelisation without modification. Some devices, however, will require additional code to maintain functionality and prevent errors when running with MPI. In order to make device development for Beatbox as approachable as possible, with as much complex, dangerous or repetitious code as possible consolidated in to reusable preprocessor macros.

3.4.1 Swapping Halos

Any device whose Run function involves reading data from neighbouring points will require data in its halos to be up-to-date. As discussed in Section 3.3.1, a device may set its `sync` field to `1` to indicate that a it requires a halo exchange to take place. To simplify implementation for third party developers, and to improve the legibility of code, *device.h* defines the `DEVICE_REQUIRES_SYNC` macro that, when included in the device's Create function, will set its `sync` field to `1`. `DEVICE_REQUIRES_SYNC` is one of a set of 'literate' macros available to device developers, all of which are discussed in Section 3.4.2.

3.4.2 Literate Macros

To simplify the development of third-party devices and to improve the legibility of their code, Beatbox provides a set of macros, defined in *device.h*, that can be used to set properties or enforce common constraints on a device's operation. The macros are intended to be 'literate', such that each macro makes a clear, declarative statement about the device's behaviour. Each of the macros is discussed below. Code excerpts showing the macros' definitions are given in Listings C.6–C.11.

`DEVICE_REQUIRES_SYNC` sets the `sync` field of the `Device` structure to `1`, ensuring that `haloSwap()` will be called in the device's Run function. In the sequential version of Beatbox, where synchronisation does not occur, this macro is conditionally compiled to be empty.

`DEVICE_ALWAYS_RUNS` sets the `alwaysRun` field of the `Device` structure to `1`, ensuring that, when its condition is `1`, the device will run in every process, regardless of its local space. This macro is used by devices that update the values of global k_variables, to ensure that the variable holds the same value in every process. In the sequential version of Beatbox, where only one process exists, this macro is conditionally compiled to be empty.

`DEVICE_IS_SPACELESS` is used to indicate that the generic space conditions (`x0`,`x1`,`y0`,`y1`,`z0` and `z1`) are not appropriate for the device. The expanded macro tests for the existence of these parameters in its parameter string and warns the user that any such value will be ignored.

`DEVICE_HAS_DEFAULT_SPACE` allows a device to insist on having default values for `x0`,`x1`,`y0`,`y1`,`z0` and `z1`. If the user supplies values for any of these parameters, an error is raised and Beatbox quits, instructing the user to remove any such parameter values.

`DEVICE_MUST_BE_NOWHERE` allows a device to insist that its `nowhere` parameter is set to `1`. If not, an error is raised and Beatbox quits.

`DEVICE_OPERATES_ON_A_SINGLE_POINT` can be used by devices whose spaces are always a single point. Since ranges of space parameters (e.g. `x0`–`x1`) are not appropriate for these devices, the user is warned that values for `x1`, `y1` and `z1` will be ignored.

3.5 Boundary Conditions

Early versions of Beatbox employed parallelised versions of the `neum*`, `dirich*` and `tor*` boundary conditions devices, which operated much like their sequential counterparts. However, when looking at existing QUI simulation scripts, two common traits were found; all simulations used Neumann boundary conditions; and all simulations placed the boundary conditions device immediately before a `diff*` device. The boundary conditions and diffusion devices are usually the only devices to require a halo swap and, when placed one before the other, the `diff*` device's haloswap is redundant. To remove the unnecessary halo exchange, and to simplify user scripts, Beatbox consolidates QUI's `diff1d`, `diff2d` and `diff3d` devices into a single `diff` device, which also enforces Neumann boundary conditions. As a result of this change, Dirichlet and toroidal conditions are unavailable in Beatbox.

The unified `diff` device detects the dimensionality of the mesh and switches between algorithms from `diff1d`, `diff2d` and `diff3d` as appropriate. To enforce boundary conditions, `diff` adjusts the computation of the Laplacian to exclude points outside the default space. Code for the unified `diff` device's run function is shown in Listing C.12. The device is developed further for use with anatomically realistic tissue geometry in Chapter 4.

3.6 The `k_func` Device

The flexibility afforded to users by the `k_func` device presents several challenges to its parallelisation. The code specified by the user in the `pgm` parameter, 'the program', will be run in each process and may affect the value of variables in the simulation medium and *global* k_variables. As such, it is necessary to ensure that the code executed by `k_func` will produce consistent, predictable results when run sequentially, and with any configuration of MPI processes. What follows is a high-level discussion of the potential pitfalls of running `k_func` in parallel, and their solutions. Details of the completed implementation follow in Section 3.6.3.

When running in parallel, each instance of a `k_func` device will iterate over points of the medium within its local space. The code specified by the user in the `pgm` block is executed once at every point in the local space. At any point in time, each of the `k_func` instances will be evaluating different points in the medium. The parallel implementation must reconcile this fact with the assumption that k_variables, which are global for the simulation, will have the same value in all processes.

3.6.1 Conditions for Safety

A `k_func` program will reliably produce the same results sequentially and in parallel if the following conditions are met:

1. no local value is assigned to a global variable;
2. no references are made to points beyond $\{x \pm 1, y \pm 1, z \pm 1\}$;

3. operations performed on local data are commutative.

These are explained in greater detail below.

3.6.2 Restricting k_func for Parallel Safety

Local-to-Global Assignment

The first problem caused by k_func programs is that it is possible for assignments to be made to global variables based on local values, as in `foo = u0;`. Since u0 can have a different value in each process, it is possible for the value of foo to differ between processes. The second problem relates to the iterative nature of k_func. Since the code in the pgm block will be executed once at each point in the device's space, incremental changes to global variables may result in different values due to the varying number of points in the local spaces of k_func instances. For example, the line `bar = bar + 1;` would result in bar being incremented by the number of points in the local space of each process. Inconsistent values in global variables may cause problems if those variables are used as the Condition of other devices, or in modifying any other behaviour that affects more than one process.

To prevent these problems from occurring, the parallelised k_func device does not allow assignments to be made from local values to global variables. To prevent problems with incrementing, assignments to global variables may only be made when the device's space is nowhere. Conversely, assignments to local variables may only be made when the device has a space. The rationale for the safety of this restriction is as follows:

1. All global k_variables will receive the same initial value, as they all read the same script file.

2. Assignments to global variables are dependent only on other global variables, which, from (1), are guaranteed to have the same value in each process.

3. Assignments to global variables are deterministic, i.e. calls to the random number function, rnd(), are prohibited. Thus, all processes will assign the same value.

4. All assignments to global variables are made on every active process.

In practice, as different processes execute the k_func program at different speeds, there may be short periods of real-time in which a global variable has different values on each process. Since no interaction between processes takes place during the execution of a k_func program and an MPI_Barrier() call ensures synchronisation after the Run function completes, any temporary differences in global variable values are entirely safe.

The Nowhere Parameter

In QUI, a device that did not operate on the simulation medium was, by convention, marked by giving it a space of `x0 = x1 = y0 = y1 = z0 = z1 = 0`. To use this set of parameters as the basis of preventing local-to-global assignments would cause problems in zero-dimensional simulations, where they are identical to those of the default space. The restrictions placed on k_func programs are such that no assignments could be made to any local variables in a zero-dimensional simulation, since all k_func instances will be taken to be nowhere.

As discussed in discussed in Section 3.2.3, the addition of nowhere as a separate space parameter disambiguates these distinct intentions. A device that should not operate on the simulation medium will have nowhere=1. The parallelised k_func device will only permit assignment to

global variables where its `nowhere` parameter is equal to `1`, and to local variables where its `nowhere` parameter is equal to `0`.

Temporary Variables

The sequential version of `k_func` permitted global variables to be used to hold temporary, intermediate values. By preventing the assignment of local values to global variables, users are forced to write long computations on a single line. To assist users, the parallel `k_func` allows *local* variables to be defined using the `def` keyword inside the `pgm` codeblock. These variables may then be used as desired. The addition of local variables raises a further problem, referred to here as the *movable value problem*.

The *movable value problem* occurs as `k_func` iterates over its space. If a local variable is able to maintain its value from one iteration to the next, it may be used to *move* a value from one location of the medium to another. The example below shows how the value of `u0` at $\{2, 4, 6\}$ could be *moved* to $\{4, 7, 9\}$ using the local variable `foo`.

```
def real foo;
foo = if(eq(x,2)*eq(y,4)*eq(z,6), u0, foo); // Pick up value
u0 = if(eq(x,4)*eq(y,7)*eq(z,9), foo, u0); // Put down value
```

Such an operation makes an assumption about the order of execution, i.e. that $\{2, 4, 6\}$ will come before $\{3, 7, 9\}$. It also assumes that the same `k_func` instance will both *pick up* and *put down* the value. In a parallel implementation, neither of these assumptions are safe. To prevent inconsistent behaviour being caused in parallel, `k_func` adds a further restriction that all variables defined inside the `pgm` block are initialised before use. Specifically, they must be given a value when defined — e.g. `def real bar 5.7;` — or appear on the left hand side of an expression *before* appearing on the right hand side.

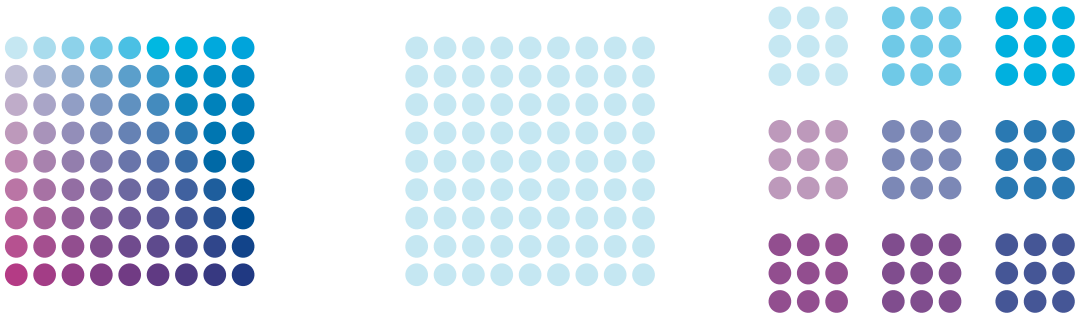
Dangerous Functions

Several functions defined in (*qpp.c*) also present problems when run in a `k_func` program. The `u()` function returns the value at a given location in `New`. In parallel, it is possible — indeed likely — that the point requested will not exist in the local subdomain. While parallelising such a function is possible, it would require a significant increase in complexity to ensure that the correct time-step was read from and to coordinate fetching the value from another process, which may not be running the function. The solution employed here avoids such complexity by removing the `u()` and three functions dependent on it (`dudx()`, `dudy()` and `dudz()`) altogether. The functionality provided by `u()` may be replicated by using the `sample` device (see Section 3.7).

In addition, the random number generation function, `rnd()`, is likely to cause problems if assigned to a global variable, as each process is highly likely to produce a unique value. By treating `rnd()` as local, assignments are restricted to local variables.

Neighbour References

One possible solution to the loss of flexibility caused by removing `u()` and the `dud*()` functions is to introduce *neighbour references*, i.e. read-only values from immediately neighbouring points. These have not been implemented in the parallel version of `k_func`, as they may introduce non-commutative operations. For example `u0 = ux0;`, where `ux0` is the value of `u0` at a neighbouring point, will effectively 'daisy-chain' the value of `u0` across the mesh. In parallel, this may produce different results, dependent on the first value of `ux0` in each subdomain.



(a) Initial conditions. (b) Sequential result. (c) Parallel result.

Figure 3.13: Inconsistency caused by daisy-chaining neighbour references. Running the `k_func` program `u0=ux0;` results in the first value in the device's space being copied throughout. In parallel, the first value in the *local* space is copied.

Aggregate Operations

The QUI version of `k_func` made it possible to aggregate local data, e.g. obtaining the sum of transmembrane voltage from across the mesh. Operations of this kind are now prevented in `k_func` programs by restrictions on assignments to global variables. The ability to perform commutative operations across regions of the mesh (summation, product, and finding minima and maxima) are made possible using the `reduce` device, described in Section 3.8.

Summary of `k_func` Modifications

Solutions to the problems discussed above can be summarised in four rules, listed below.

- The LHS may only be a global variable when `nowhere` is TRUE (`1`).
- The RHS may only be a local variable when `nowhere` is FALSE (`0`).
- Locally defined variables must be initialised before use:
  - e.g. `def real foo; u0 = 3 * foo;` is illegal,
  - as is `def real foo; def real bar foo;`.
- Global variables must not be assigned values from local symbols.

3.6.3 Parallel Implementation

This section discusses the modifications made to the `k_func` device in order to address the issues discussed above. For a discussion of the sequential operation of the device, see Section 2.11.

`k_func` copies the default symbol table, `deftb`, to its local symbol table, `loctb`. Local symbols, such as `u0` are inserted into `loctb` only. The parallel version of `k_func` uses this fact to identify symbols as local or global. Global symbols will compile against `deftb`, while local symbols will not.

To identify local variables that are used before being initialised, `k_func` maintains a list of uninitialised local variables. The list is added to when a new local variable is defined without an initialiser. A variable is removed from the list when it is used on the right hand side of a statement.

Making `rnd()` Local

To prevent `rnd()` from being assigned to a global variable, the function is defined as a local symbol. `rnd()` is now inserted into `loctb` in the `k_func` device's Create function.

Analysing the Program

The majority of the adaptations made to the `k_func` device for parallel operation are implemented as changes to *k_comp.h*, the file responsible for the parsing and compilation of the `pgm` block. To cover each of the possible eventualities described by the rules above, the flow of execution in *k_comp.h* must follow the decision tree shown in Figure 3.14.

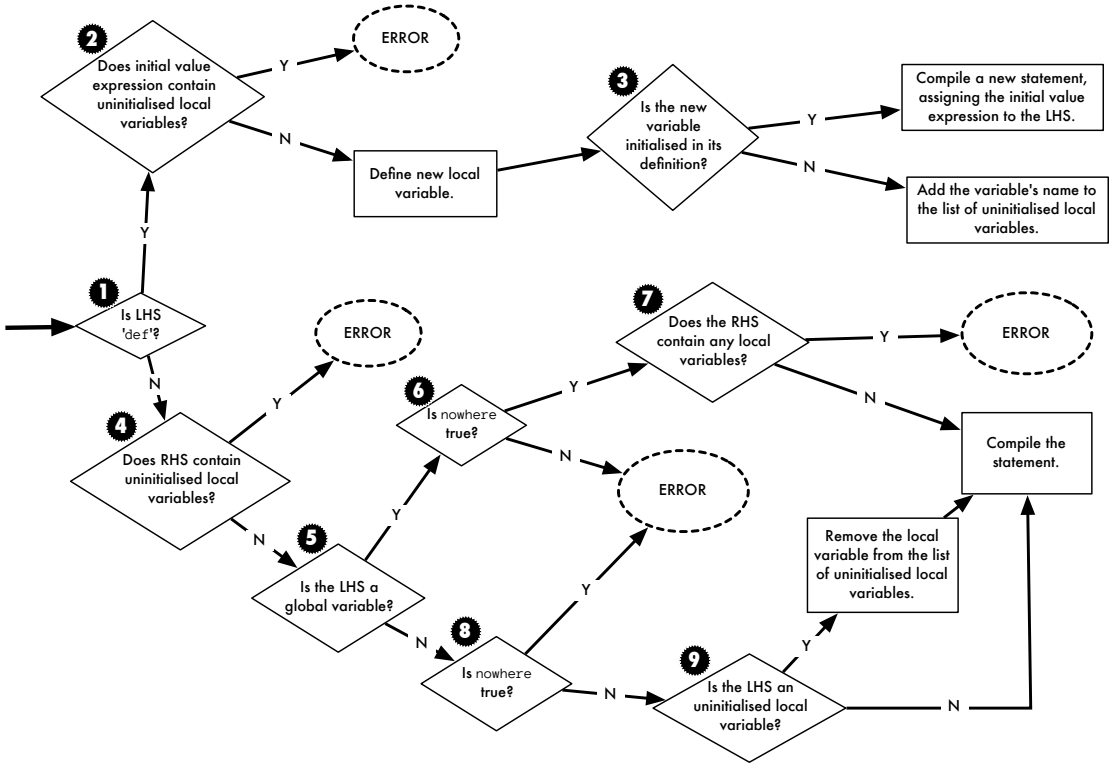


Figure 3.14: *k_comp.h* Decision Tree. Flow of control when parsing `k_func` programs.

Read-Only Variables

As discussed above, the sequential `k_func` device allowed assignments to be made freely to any k_variable in `deftb`. While this flexibility is broadly safe in the context of a sequential simulation, there are a number of k_variables whose values should remain unchanged throughout the simulation, regardless of the underlying hardware implementation. The current timestep, `t` should reasonably be expected to increase once for every iteration over the ring of devices. For `t` to be assigned an absolute value, without validation, would violate the calculations in some RHS modules, and may lead to other unexpected behaviour that could confuse users. Similarly, to adjust the values of the simulation medium's dimensions (`xmax`, `ymax`, `zmax` and `vmax`) would

not vary the memory allocated to `New`, but change devices' view of the data therein, jeopardising the integrity of calculations and the safety of the simulation run.

To prevent the assignment of values to certain variables, while still allowing access to their values, a new class of *read-only* variable is added. The status of the variable is checked in *k_comp.h* when reading the left hand side of a `k_func` program statement.

Variables defined as read-only are listed below:

`t` The current timestep.

`inf` Infinity (equivalent to `LONG_MAX` in C).

`xmax` Number of points in the x-axis.

`ymax` Number of points in the y-axis.

`zmax` Number of points in the z-axis.

`pi` π.

`always` Default device `Condition`. Equals `1`.

3.6.4 Limitations of the Parallelised `k_func` Device

The restrictions imposed on the `k_func` device, as described above, effectively divide local and global operations and two classes of operation — *reduction* and *selection* — are prevented entirely. Reduction refers to producing a single value from many, for example summing. Selection refers to the process of identifying a single value by its absolute coordinates in the medium, as performed by `u()`. To maintain this useful functionality whilst ensuring safety in parallel, two new devices — `sample` and `reduce` have been introduced. Both devices are described in detail below.

3.7 The `sample` Device

The `sample` device provides basic selection functionality, by allowing the user to assign an arbitrary value from the simulation medium to a global k_variable. The assignment is made on all active processes, and consequently is safe when run in parallel.

The device works by identifying the process that 'owns' the point to be sampled. When the device is run, this process broadcasts the sampled value to all other processes, who in turn assign the value to the desired k_variable.

3.7.1 Creating `sample`

Because the sampled value must be assigned to the selected variable on all processes, the Create function includes the `DEVICE_ALWAYS_RUNS` macro. This ensures that, regardless of the device's space, every instances will run in order to make the assignment. Secondly, because the device only samples a single value from the simulation medium, the `DEVICE_OPERATES_ON_A_SINGLE_POINT` macro is used. The device accepts two device-specific parameters, `result`, the `real` k_variable to which the sampled value should be assigned and `debug`, the optional file to which the sampled value can be printed. The rank of the 'root' process is determined by calling `getRankContainingPoint()`. To prevent duplicate debug output, only the root process will accept the `debug` parameter and open the corresponding file. Code for the `sample` device's Create function is given in Listing C.15.

Accepting k_variables

Although the process of accepting a k_variable as a parameter is very similar to that performed by `accept_condition()` (*qpp.c*), QUI had no pre-defined mechanism for doing so. For that reason, the MPI implementation of Beatbox adds the `accept_real_variable()` function and corresponding `ACCEPTV()` macro to *qpp.c* and *qpp.h* respectively. Listings C.13 and C.14 show the code in detail.

3.7.2 Running `sample`

When run, the root process will assign the sampled value to its instance of the `result` variable, and print the value to the debug file, if present. The value will then be broadcasted to all active processes using the `MPI_Bcast()` function. On processes other than the root, this value is received directly into the `result` variable.

3.8 The `reduce` device

The `reduce` device is added to the MPI implementation to allow users to carry out calculations over the whole simulation medium, or any subset thereof. When run, `reduce` performs one of a set of commutative operations — sum, product, minimum or maximum — over all of the points within its *global* space, assigning the result to a user-specified (global) k_variable.

When run sequentially, the `reduce` device performs a simple iterative function over the simulation medium to find the sum, product, minimum or maximum. The value can then be directly assigned to the specified k_variable. In parallel, the device instance in each process performs the same iterative function over its local space. Having arrived at a *local* sum, product, minimum or maximum, the values from each process are then reduced further using an `MPI_Reduce()` function call. The result of the global reduction is finally broadcast to all processes, to be assigned to the desired k_variable.

`reduce` operates on two MPI communicators: the reduction operation takes place on all active instances of the device, i.e. those where the device space intersects with the local subdomain; the broadcast of the final value takes place on all active processes.

3.8.1 Creating `reduce`

The `reduce` device's create function accepts two device-specific parameters; `operation` and `result`; both of which are compulsory. `operation` is a string indicating the reduction operation to be performed over the device's space. It may have one of four values: `sum`, `prod`, `min` or `max`. When a valid selection is provided, the corresponding *local* reduction function — `reduce_sum()`, `reduce_prod()`, `reduce_min()` or `reduce_max()` — is assigned to the `local_reduce` field of the device's parameter structure. In the MPI implementation, the `operation` parameter also selects a corresponding `MPI_Op` operator — `MPI_SUM`, `MPI_PROD`, `MPI_MIN` or `MPI_MAX` — to be used when collecting the results from all processes. The MPI operation is assigned to the `global_reduce` field of the parameter structure. The `result` parameter is a k_variable of type `real` to which the result of the operation should be assigned, and is accepted using the `ACCEPTV()` macro.

Finally, the MPI implementation defines a communicator on the active instances of the device using `deviceCommunicatorWithFirstRank()`. When run with MPI, the device operates on two communicators: the reduction operation takes place on a device communicator defined on the device's space, while the assignment of the final result must be distributed to all active processes using the `ALL_ACTIVE_PROCS` communicator. Collective communication on `ALL_ACTIVE_PROCS`

will require that all instances of the device, even those outside its space, are running. This is achieved by using the DEVICE_ALWAYS_RUNS macro in the Create function.

Code for the reduce device's run function is given in Listing C.17.

3.8.2 Running reduce

When run, the reduce device iterates over its space, running the function pointed to by the reduce_local field of the parameter structure. The local reduce function will update the local variable running accordingly, such that it maintains the current sum, product, minimum or maximum. If running sequentially, the value of running on exiting the loop can be immediately assigned to result. In MPI, however, running currently contains the sum, product, minimum or maximum for its *local* space only. To collect and further reduce the value of running on each process, the MPI_Reduce() function is used. Using the communicator defined in the Create function MPI_Reduce() will cause the root process to collect the value of running from all processes in the communicator. These values will be further reduced using the corresponding MPI operation and ultimately assigned to result on the root process.

The desired value for result now resides on the root process only. This value must now be distributed to *all* processes, not just those in the device's communicator. This is achieved by distributing the value of result to all processes in the ALL_ACTIVE_PROCS communicator using MPI_Bcast().

Code for the reduce device's run function is given in Listing C.18.

3.9 The poincare Device

QUI provides a device called k_poincare, that can be used to watch a value execute a user-defined function if the value crosses a given threshold. This combination of functionality overlaps with k_func. Indeed, it is possible to implement k_poincare using k_func devices. As a result, many of the parallelisation issues caused by k_func arise with k_poincare.

To prevent unnecessary complexity in the code, and to retain the ethos of devices that do *one thing, well*, Beatbox replaces the k_poincare device with poincare. The poincare device can 'watch' the value of any value in the simulation medium and detect crossings in time of a given threshold value. The user may choose to detect only positive crossings, negative crossings, or both. If a crossing in the desired direction is detected, the value of a user-specified k_variable will be set to 1. If users wish a crossing to trigger the execution of a user-defined function, this k_variable can be used as the Condition of a k_func device. Optionally, the timestep on which the last crossing occurred can be assigned to a second user-specified k_variable.

3.9.1 Creating poincare

poincare accepts four device-specific parameters. cross is a constant of type real that specifies the threshold about which a crossing will be detected. sign is an integer that indicates the direction of crossings that will be detected. A value of -1 will detect negative crossings only, 1 will cause only positive crossings to be detected and 0 will cause crossings in either direction to be detected. No other values for sign will be accepted. result is the k_variable of type real that will be used to indicate whether or not a crossing has been detected this timestep. timestep is an optional parameter, specifying a k_variable of type real that will hold the timestep on which the last crossing occurred.

The poincare device is similar in operation to sample, in that it assigns the result of an operation from one point in the mesh to a k_variable on every process. As in sample, the

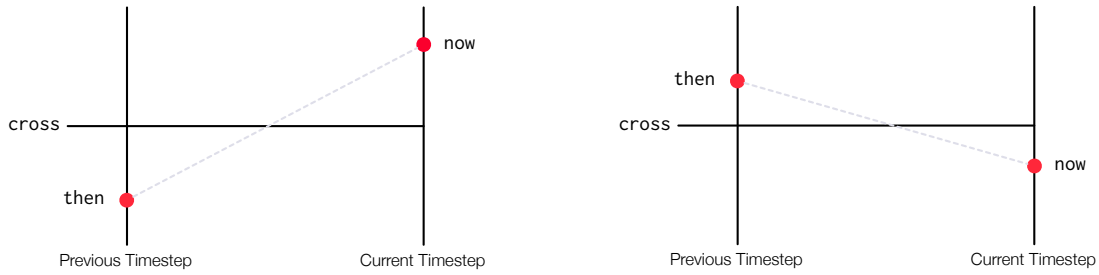
`DEVICE_ALWAYS_RUNS` macro is used to ensure that the value of the result variable in every process is the same. Also, as only one point in the mesh can be watched, `DEVICE_OPERATES_ON_A_SINGLE_POINT` is used. The Create function identifies the process in which the 'watched' point resides. The rank of this process is held in the `root` field of the parameter structure. The `first` field of the parameter structure is set to `1` to indicate that the device has not yet been run. This will prevent the erroneous detection of crossings when first run.

Code for the `poincare` device's Create function is given in Listing C.19.

3.9.2 Running `Poincare`

DetectCrossing()

```
1  now ← New[x, y, z, v]
2  if first = 1
3     then
4          first ← 0
5     else
          ▷ Negative crossing
6         if sign ≠ Positive and (then > cross and cross > now) or
          ▷ Positive crossing
          sign ≠ Negative and (then < cross and cross < now)
7            then
8                 crossingDetected = 1
9                 timestep = t
```



(a) Positive Crossing (b) Negative Crossing

Figure 3.15: Crossings in the `poincare` device. A crossing occurs when the values of `then` and `now` straddle that of `cross`. A positive crossing occurs when `now` is greater than `then`.

When run in parallel, the root process samples the specified value, `now`, and compares it to the previously sampled value, `then`. If `now` and `then` straddle the value supplied to `cross`, and the direction of the crossing is allowable with respect to `sign`, the local instance of the `result` variable is set to `1` and, if present, the local instance of the `timestep` variable is assigned the current timestep. Pseudocode for the procedure is shown in DetectCrossing. Figure 3.15 illustrates the definitions of positive and negative crossings.

The value of `result` and `timestep` are then placed into a small buffer array before being broadcast to all active processes. Receiving processes extract the values from the array and assign them to their local instances of `result` and `timestep`. Finally, `now` is assigned to `then`, ready for the next time `poincare` is run.

Code for the `poincare` device's Run function is given in Listing C.20.

3.10 MPI Input/Output

Input from and output to files on disk is greatly complicated by running in parallel. For each process to read and write separate files would be extremely inconvenient for the user, especially for large process counts, and would effectively fix the allowable domain decomposition for any subsequent use of the files. Wherever possible, Beatbox attempts to access a single file in parallel, with each process reading data to, or writing data from its local subdomain. Writing to a file in a sequential program involves serialising three- or four- dimensional data into a one-dimensional file, doing so in parallel requires knowing not only what to write, but which region of the file to write to. I/O is also the most time consuming of parallel tasks, as it relies heavily on collective operations, network communication and access to disk.

This section deals with the techniques used to partition files between processes and to write to or read from files in parallel. It is followed by a discussion of Beatbox I/O-capable devices: `dump`, `load`, `ctlpoint`, `ppmout` and `record`.

3.10.1 File Partitioning

In order for a file to be accessed in parallel, it is necessary to *partition* the file between the processes involved. The regions of the file that a process may access are defined by its *view*. The view is an MPI datatype, that may be either an in-built primitive (e.g. `int` or `char`) or derived. For almost all applications in Beatbox, derived types are used.

MPI treats the file as a one-dimensional array, of which each process should access a discrete region. When accessing one-dimensional data, this can be achieved by each process adjusting its view of the file by an agreed offset. When writing multi-dimensional data, however, the data written by a single process is unlikely to be contiguous in the file. To simplify partitioning, MPI provides the `MPI_Type_create_subarray()` function, which defines an MPI derived datatype that maps an n-dimensional region of an n-dimensional dataset to a one-dimensional array. The function takes the dimensions of the total volume of data to be read or written, the dimensions of the local region and their position within the whole. The result is a derived datatype that can be used as the process's view of the file. When correctly applied, the union of all processes' views will equal the total data to be read or written, without overlaps.

Subarrays can also be used to define the regions of an array in memory to be read. In Beatbox's MPI implementation, it is common to see file writing involving two MPI derived datatypes: one to select the required subset of data from `New` (usually the local space), and another to map this data into the file. For example, `dump` defines a datatype that allows it to access its local space from within `New`, and a second datatype, based on its local space within the global space, that defines its view of the file.

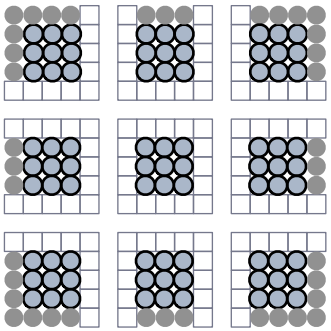
As shown in Figure 3.16, derived datatypes describe an access pattern to read subsets of n-dimensional data from a one-dimensional array.

3.10.2 File Handling

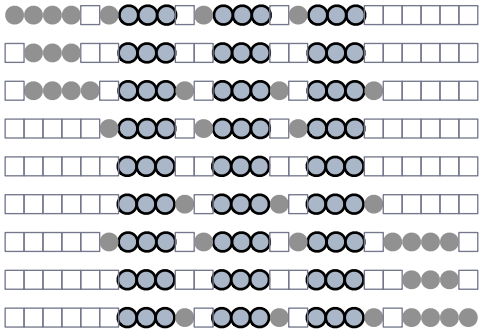
MPI provides its own set of functions to handle the opening, closing and other manipulations of files. If a file is to be accessed in parallel, it is necessary that these functions are used throughout. File handles in MPI routines are of the type `MPI_File`, requiring that declarations in device parameter structures are conditionally compiled to use the appropriate type.

The process of opening a file is collective, meaning that it must be called by every process in the MPI communicator. All subsequent collective actions on the file, including closing, involve

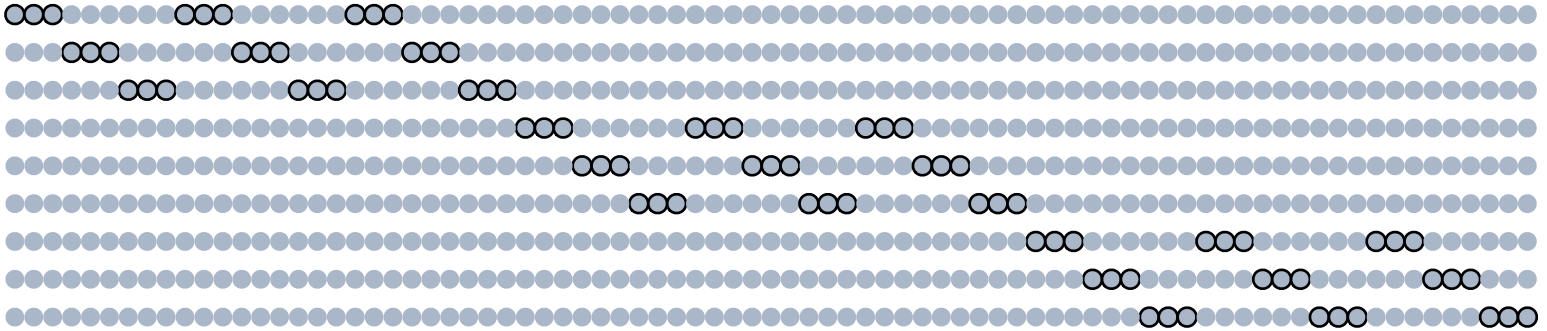
every process in the communicator. Consequently, a device should declare a communicator on its active instances to handle I/O if it is to avoid deadlock when accessing the file collectively.



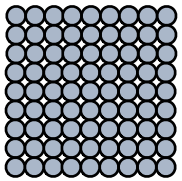
(a) The decomposed space to be written to file.



(b) Source types for each process, defining the local space as part of `New`.



(c) Filetypes for each process, defining the local space as part of the global space.



(d) The completed file.

Figure 3.16: Mapping source types to file types. For each process, points at locations marked by black circles are read from `New` using the source type, and written to the location marked by next black circle in the process's file type.

3.11 The `dump` and `load` Devices

`dump` writes a four-dimensional hypercube of binary data, from its space in `New` to a file. The device defines two derived datatypes — `filetype` defines the device's view of the file, while `sourceType` defines the data to be accessed from `New`. The code used to define `filetype` and `sourceType` is given in Listing C.21. Creation of a communicator for file access is handled by calling `deviceCommunicator()`. Code for the `dump` device's Create function is given in Listing C.22.

When run, `dump` uses `MPI_File_write()`, to write one instance of its `sourceType` to the `filetype`. This will write its local space to the file so that, when all active processes have written their part, the file will resemble the one created sequentially. As can be seen from Listing C.23, the sequential branch of the Run function iterates over the local space, writing each point using `fwrite()`.

The `load` device performs precisely the opposite function to `dump`. Where dump writes the contents of a subset of the simulation medium to file, `load` updates a subset of the simulation medium with the values read from a file. For this reason, the `load` device uses the same derived datatypes defined in *dump_types.h*. When reading the file, `load` uses the `MPI_File_read()` function, reading *from* `filetype` *to* `sourceType` to map values from the file to its local subdomain in `New`. Code for the `load` device's run function is given in Listing C.24.

3.12 The `ctlpoint` Device

The `ctlpoint` device reads and writes a file containing a *control point*, a snapshot of the entire simulation state. This can be used to guard against system crashes, allowing the simulation to be restored to the last control point. Control points are also useful when running a set of simulations that repeat the same initial period before diverging; by capturing a control point from which each divergent path can start, repetition of the initial phase can be avoided. Each control point comprises three sets of data:

Simulation properties Version name of the software and the dimensions of the mesh.

Simulation state State of the entire simulation medium, in a format identical to that used by the `dump` device.

k_variables The name, type and value of every `k_` variable.

3.12.1 Creating the `ctlpoint` Device

The Create function for the sequential `ctlpoint` device accepts four script parameters. `file` (`str`) is the relative path to the file to be used. `enrich` is an `int` 'boolean', used to select whether the control point file can be used to add `k_` variables to a simulation (described below). `rkv` is an `int` boolean used to indicate whether the file should be archived using the UNIX `rkv` tool. Finally, `debug` is a relative path to an optional debug file, to which verbose messages will be written.

The state of the simulation can be written using the derived datatypes defined in *dump_types.h*. To ensure that the entire medium, including absolute boundaries is written to file, the device's global space is fixed to `0`–`max-1` on every axis. Local spaces are adjusted accordingly to include internal points and absolute boundary points only. `runHere` is also set to `1` on all devices, as data from every process must be backed up and restored. An MPI derived datatype, `k_var_Type`, is defined to represent k_variables, with fields for their type, name and value.

Code for the ctlpoint device's Create function is given in Listing C.25.

3.12.2 Running ctlpoint

When first run in a simulation, ctlpoint will attempt to read from file, if one exists. At every subsequent run, ctlpoint writes to file. When writing to file, ctlpoint creates a temporary file, whose name is suffixed with ~. Once saving is complete, the temporary file is renamed to replace the existing file.

Reading a control point file begins by reading simulation properties. Each of the properties to be read is listed in a line of *ctlpoint.h* using the D() function macro.

D(name,new,old,type,size,erraction) expands to read size variables of type from the file to the address specified in new, compare them with the value pointed to by old. If different, erraction is performed. Due to the distinction between C and MPI datatypes (e.g. char and MPI_CHAR), the MPI implementation adds the CHAR_VAR and INT_VAR as alternatives to D for char and int variables respectively. Unlike the sequential version, the MPI implementation does not treat the mesh as an additional simulation property. Instead, the mesh is explicitly read from a file in the same manner as load. After reading the control point, the MPI version of ctlpoint calls haloSwap(). This ensures that internal boundaries are updated, since they are not written to the control point and may be required by subsequent devices. While devices that use internal boundaries should call haloSwap() themselves, this call provides extra safety. Halo swapping is idempotent, and has no effect on the correctness of the equations — internal boundaries can never be too up-to-date.

The third phase of reading the control point is to read k_ variables. The MPI implementation uses the k_var_Type derived datatype to read from file. The name of each variable read is checked against the default symbol table. If the symbol already exists in the table, its type read from the file must match the existing definition. If the symbol does not already exist, the enrich parameter, indicates whether a new variable can be declared. If an unrecognised variable is read, and enrich equates to FALSE, an error is raised and the simulation will quit. Finally, the value of the read variable is assigned to the variable in the symbol table.

The process of writing a control point is similar to that of reading one; simulation properties, the simulation medium and k_ variables are each written to the file in turn. The key difference is that simulation properties and k_variables are written by process 0 only. This can be done safely, since the values of system properties and k_ variables can be assumed to be the same on all processes. The simulation medium is written in parallel in a manner similar to dump.

Code for the ctlpoint device's MPI Run function is given in Listing C.26.

3.13 The ppmout Device

Parallelising the ppmout device brings with it some additional challenges, in the form of *sequences*. Sequences allow a device to access a sequentially numbered set of files, without having to explicitly create and open each file.

3.13.1 Sequences In MPI

The sequence Datatype

Since the sequence datatype (*sequence.h*) refers to a file, it is necessary to provide a file handle of type MPI_File when run with MPI. Also, as the file access mode is user-specified, the mpiMode

field is added to store the equivalent MPI file access mode as an integer. Finally, a communicator, `comm`, defines the group of processes that will operate on the files accessed by the sequence.

Creating a `sequence` under MPI

The process of creating a sequence is altered under MPI such that active devices will open files using MPI I/O functions. The MPI version of `acceptq()` function is given an added `runHere` parameter, which is used to indicate if the calling device instance is active. The calling device's `runHere` parameter is included in the MPI version of the `ACCEPTQ` macro, such that no alteration is necessary in the device code. The MPI implementation defines a communicator over active devices, which is then assigned to the `comm` field of the `sequence` structure. Unlike `fopen()`, MPI file access functions use integer constants defined in the MPI library. The `getMPIFileMode()` function is added to *sequence.c* to obtain MPI.

Code for the functions described above is shown in Listings C.27–C.30.

Advancing a `sequence` under MPI

The MPI implementation of `nextq()` is handled in much the same way as `acceptq()`, with the majority of the function's logic remaining, and MPI file access functions used where necessary.

3.13.2 Creating `ppmout`

After accepting its parameters, which are unchanged from QUI, the MPI implementation defines a derived datatype with which to partition the PPM file. As discussed in Section 2.12.2, multiple z-layers of the file can be written to form a three-dimensional image. For this reason, the filetype is described as three-dimensional. Each point in the array is of type `pointType`, which itself is a derived datatype comprised of three contiguous `MPI_CHAR` types. Definitions of the derived datatypes are shown in Listing C.31.

3.13.3 Running `ppmout`

Unlike `dump`, which defines a second datatype with which to access data from `New`, `ppmout` must first map values from the points in its space to RGB colour values. To do this, `ppmout` defines and populates a four-dimensional array, `buf`, with RGB values. `buf` can then be accessed directly when writing to disk. As shown in Listing C.32, `buf` is declared using local dimensions from the device's parameter structure.

In parallel, the process at the start of the global space handles writing the file header. The view of all processes is then displaced by the length of the header, so that the image data will begin after the header, instead of overwriting it. Listing C.33 shows this in action.

Finally, the image data are buffered before being written to file in a single call to `MPI_File_write()`. With MPI I/O, writing one large block of data performs significantly better than writing one point at a time. Listing C.34 shows the parallel and sequential branches of the Run function writing data to file. The background colour feature apparent in this listing is only used with anatomically realistic geometry, and is introduced in Section 4.8.

3.14 Verification

Beatbox's MPI implementation was tested with a series of simulations, run over thousands of timesteps to allow any errors to grow. Simulations were run with and without stimulus, and

Processes	Active Processes	Decomposition
1	1	$(1 \times 1 \times 1)$
4	4	$(1 \times 2 \times 2)$
8	8	$(2 \times 2 \times 2)$
16	16	$(2 \times 2 \times 4)$
32	32	$(2 \times 4 \times 4)$
64	64	$(4 \times 4 \times 4)$
128	125	$(5 \times 5 \times 5)$
256	252	$(6 \times 6 \times 7)$

Table 3.3: Decompositions of the *bigbox* scripts.

on QUI using equivalent scripts. Automated comparisons of output files showed that Beatbox and QUI produced identical results. Further simulations compared the sequential and parallel modes of Beatbox, using varying numbers of processes and decompositions. Again, automated comparisons of output files confirmed identical results between implementations.

3.15 MPI Performance

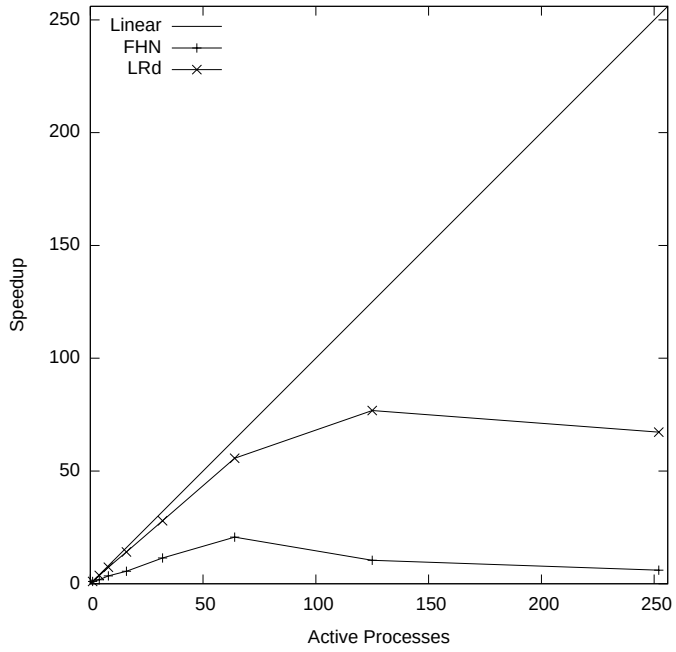
MPI performance was measured by running a simulation on a large medium, with a mesh size of $300 \times 300 \times 300$. `euler` `diff`, `dump`, `ppmout` and `record` were run every timestep. The simulation was run using FitzHugh Nagumo (FHN) and Luo Rudy (LRd) kinetics on 1,4,8,16,32,64,128 and 256 processes. The scaling performance of a single timestep was averaged over 100 timesteps. Due to cost constraints, simulations were limited to a maximum wall-clock time, adjusted to suit the parameters of the simulation. As a result, some simulations did not reach 100 timesteps. In each of these cases, it was considered that sufficiently many timesteps had been recorded — in the high tens — to provide an representative sample. Scripts used are shown in Appendix D. The wallclock time taken by each device was recorded for each timestep and the mean used to show scaling. The hardware used is described below.

The HECToR Phase 2a system comprises 3072 quad-core compute nodes, amounting to a total of 12,288 cores, each of which acts as a single CPU. The processor is an AMD 2.3 GHz Opteron. Each quad-core socket shares 8 GB of memory. Cray XT4 operating system has been upgraded to CLE 2.2. Each quad-core socket controls a Cray SeaStar2 chip router. This has 6 links which are used to implement a 3D-torus of processors. The point-to-point bandwidth is 2.17 GB/s, and the minimum bi-section bandwidth is 4.1 TB/s. The latency between two nodes is around $6\mu s$. 508 TB of high-performance RAID disks are controlled by 3 controllers through 12 IO nodes. The disks are accessible globally from any phase 2a compute node and use the Lustre distributed parallel file system [UoE HPCX Ltd., 2007–2010].

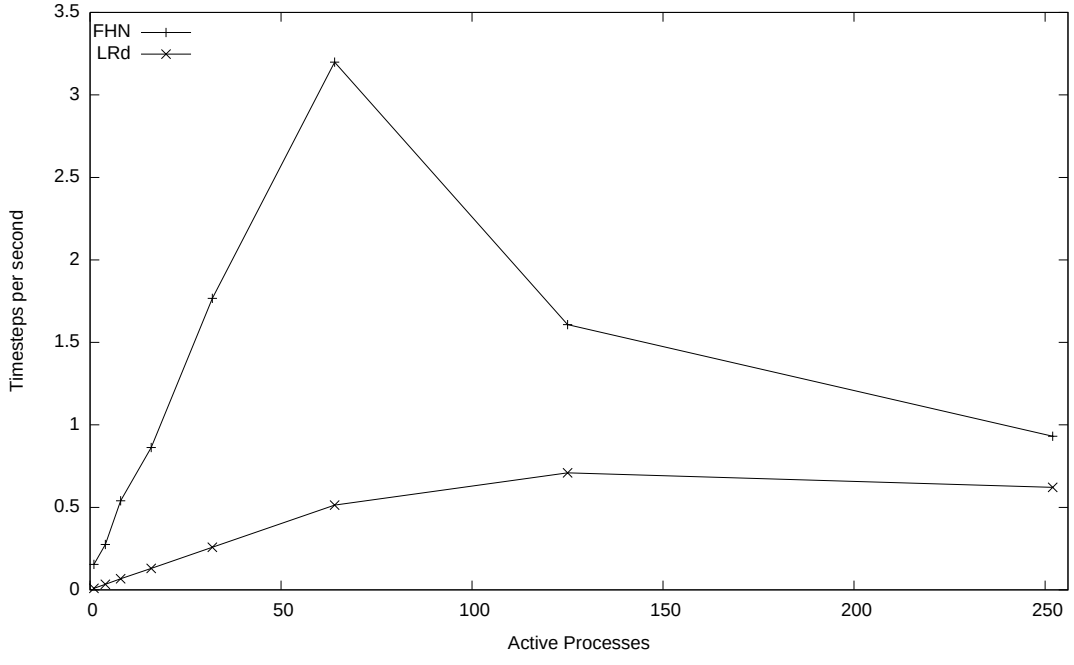
Table 3.3 details the decompositions of the mesh for each processor count.

Running every device on each timestep gives a slightly unbalanced picture of simulation scaling. Figure 3.17 gives a view of typical topline performance, by running `euler`, `diff` on every timestep, with a single I/O device, `dump`, run on every 100th timestep. It shows weak scaling performance, peaking at 64 processes for FHN kinetics, and at 128 for LRd. To more closely examine the causes of this, Figure 3.18 breaks scaling performance down by device.

As can be seen from Figure 3.18, the `euler` and `diff` devices show strong scaling performance with both models of kinetics. Other devices, however, perform far less well.

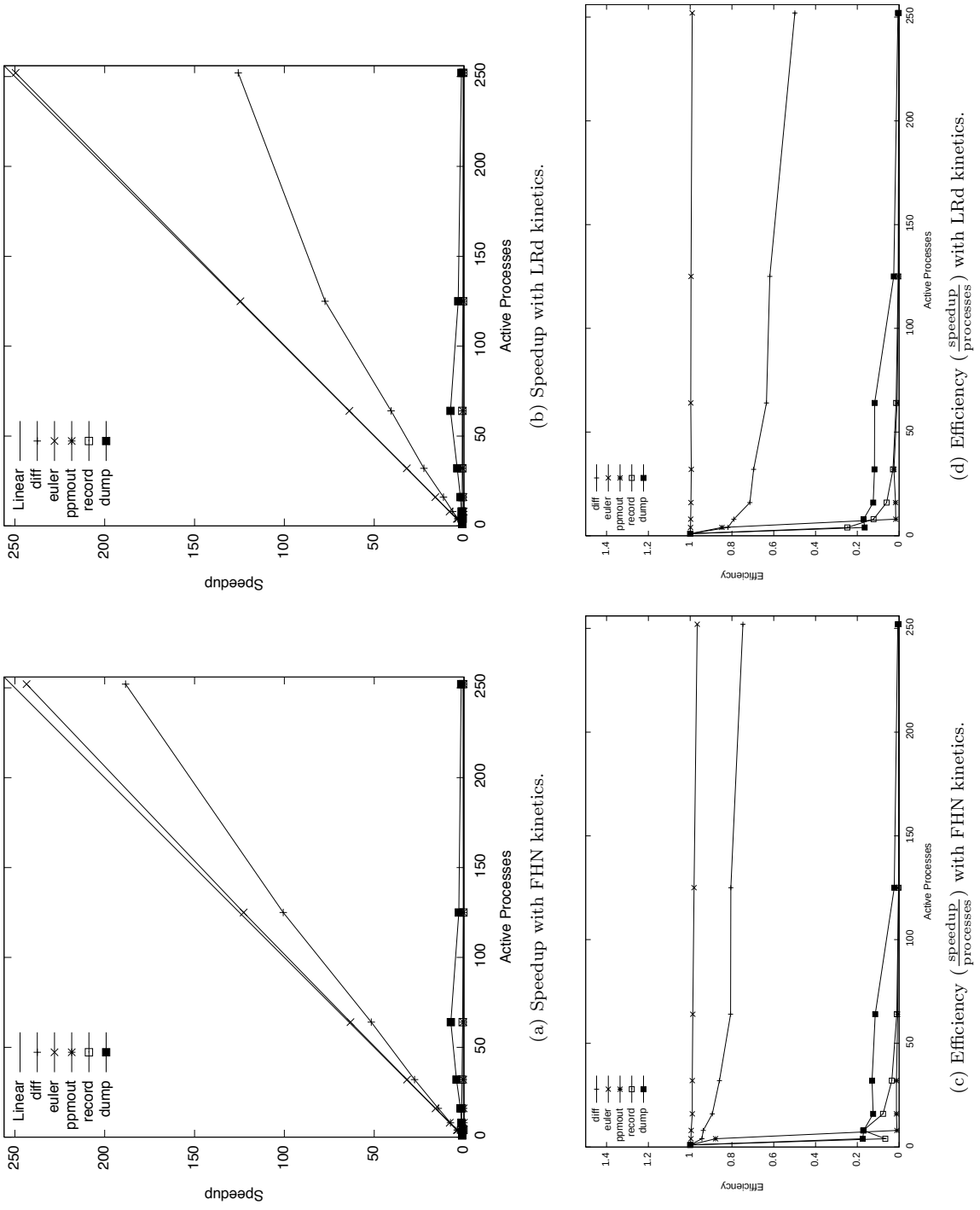


(a) Speedup



(b) Timesteps per second

Figure 3.17: Typical simulation performance, using `euler diff` on every timestep, and `dump` on each 100th.



(a) Speedup with FHN kinetics.

(b) Speedup with LRd kinetics.

(c) Efficiency ($\frac{\text{speedup}}{\text{processes}}$) with FHN kinetics.

(d) Efficiency ($\frac{\text{speedup}}{\text{processes}}$) with LRd kinetics.

Figure 3.18: Device performance.

3.15.1 `euler` Performance

As shown in Figure 3.19, `euler` scaling performance improves slightly when using LRd kinetics. This suggests that `euler` 'prefers' more computationally expensive models. This may be due to the increased, evenly balanced, workload reducing the effect of synchronisation time.

3.15.2 `diff` Performance

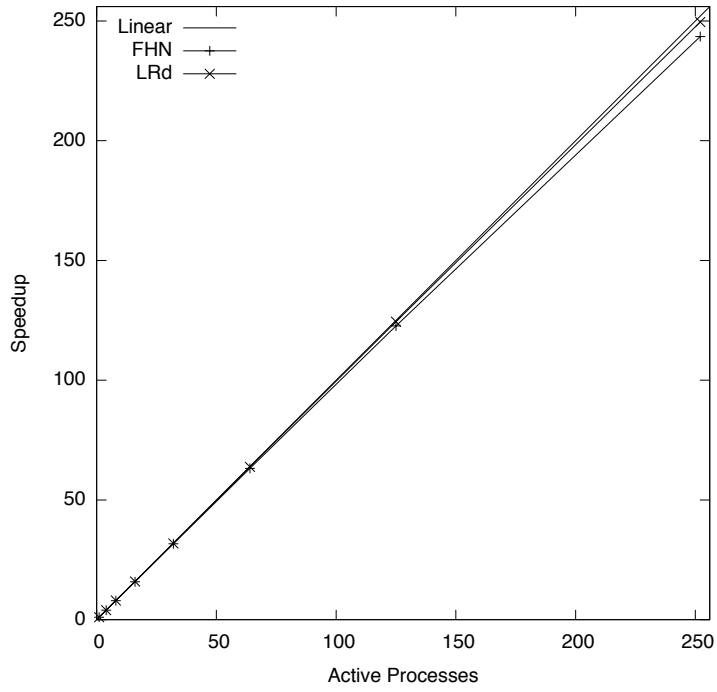
Figure 3.20 shows that `diff` scaling is adversely affected by LRd kinetics. This is likely due to the twelve-fold increase in data to be communicated during `haloSwap()`.

3.15.3 I/O Performance

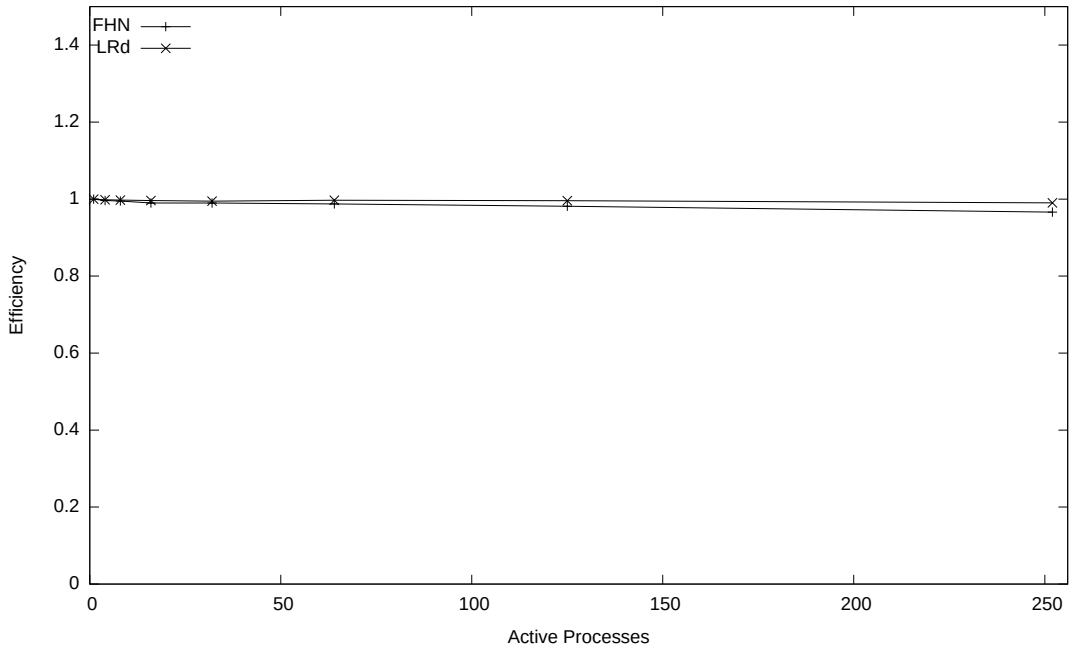
Relative to other devices, the scaling performance of I/O devices is poor. In general, Beatbox's parallel performance is bound I/O. This is in part due to the poor scalability of I/O devices, and in part to the large I/O component in the simulation tasks tested. Figure 3.22 shows the share of sequential runtime devoted to each device. More complex models, such as LRd, swing the balance of runtime in favour of computation, which scales well. Consequently, it can be expected that more complex computation will further improve Beatbox's parallel efficiency. To address the issue of I/O scaling for simulations of all kinds, it would be beneficial to fix the number of processes writing to disk at a number optimal for the machine. Instead of writing directly to disk, processes would instead communicate their data to the nearest designated 'writing process'. This approach would greatly reduce contention for file servers and would ultimately allow computation to scale freely of I/O. This approach has been used successfully by Bush et al. [2010] and Plank et al. [2010].

The approximately equal scaling performance of `dump` between FHN and LRd models (Figures 3.21a and 3.21b) shows that large ($\times 12$) increases in the total volume of data to be written do not have a significant impact on scaling. If anything, it can be expected that larger writes to disk will scale more efficiently. From this, it is reasonable to suggest that increased mesh sizes should also produce similar I/O scaling performance and benefit from further improvements in the computation-to-I/O ratio. This bodes well for studies of the kind described in [Bernabeu et al., 2008], where a mesh of $1024 \times 1024 \times 2048$ has been used. For both models, however, there is a strong peak in scaling performance at 64 processes. This phenomenon is similar to that observed in molecular dynamics code DL-POLY, running on the same machine. In Bush et al. [2010], 64 processes was found to be the optimum number of 'writing' processes for MPI I/O functions.

It is less clear why `dump` so clearly outperforms the other I/O devices. It may be that for both models, `dump` writes at least twice the volume of data as `ppmout`. Larger writes are generally found to be more efficient for MPI I/O. It may also help that `dump` does not buffer its output, while `ppmout` and `record` both do.

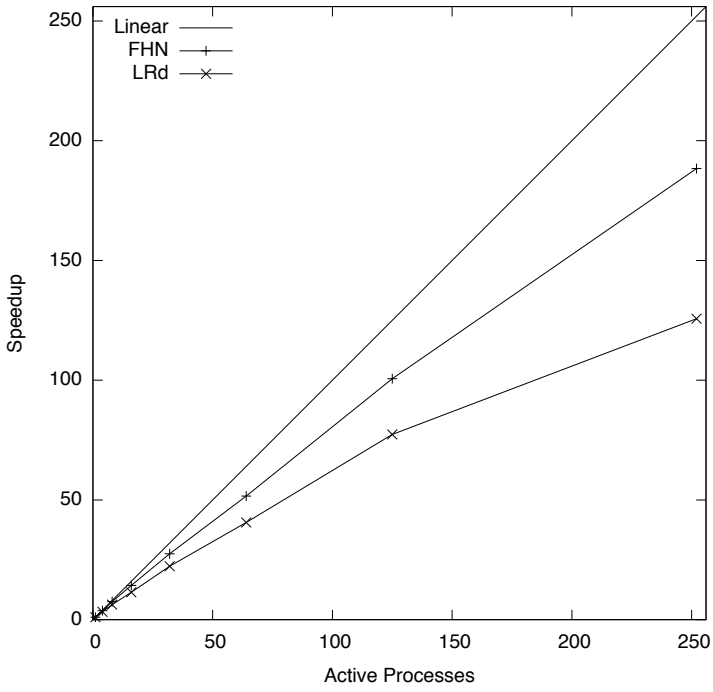


(a) Speedup

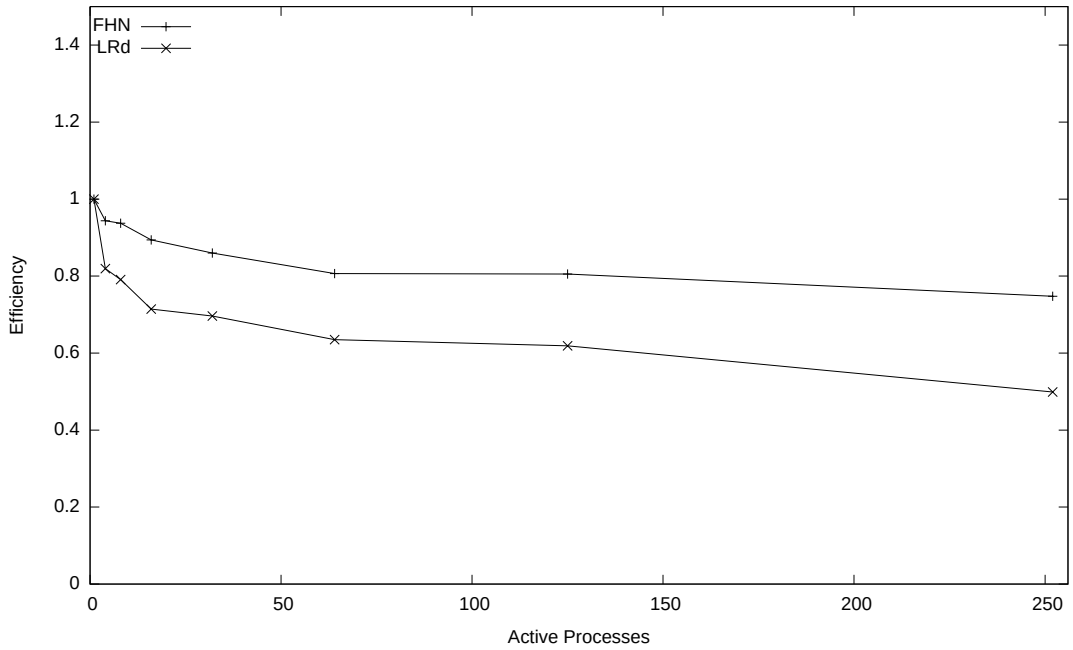


(b) Efficiency

Figure 3.19: Comparison of `euler` performance using FHN and LRd kinetics.

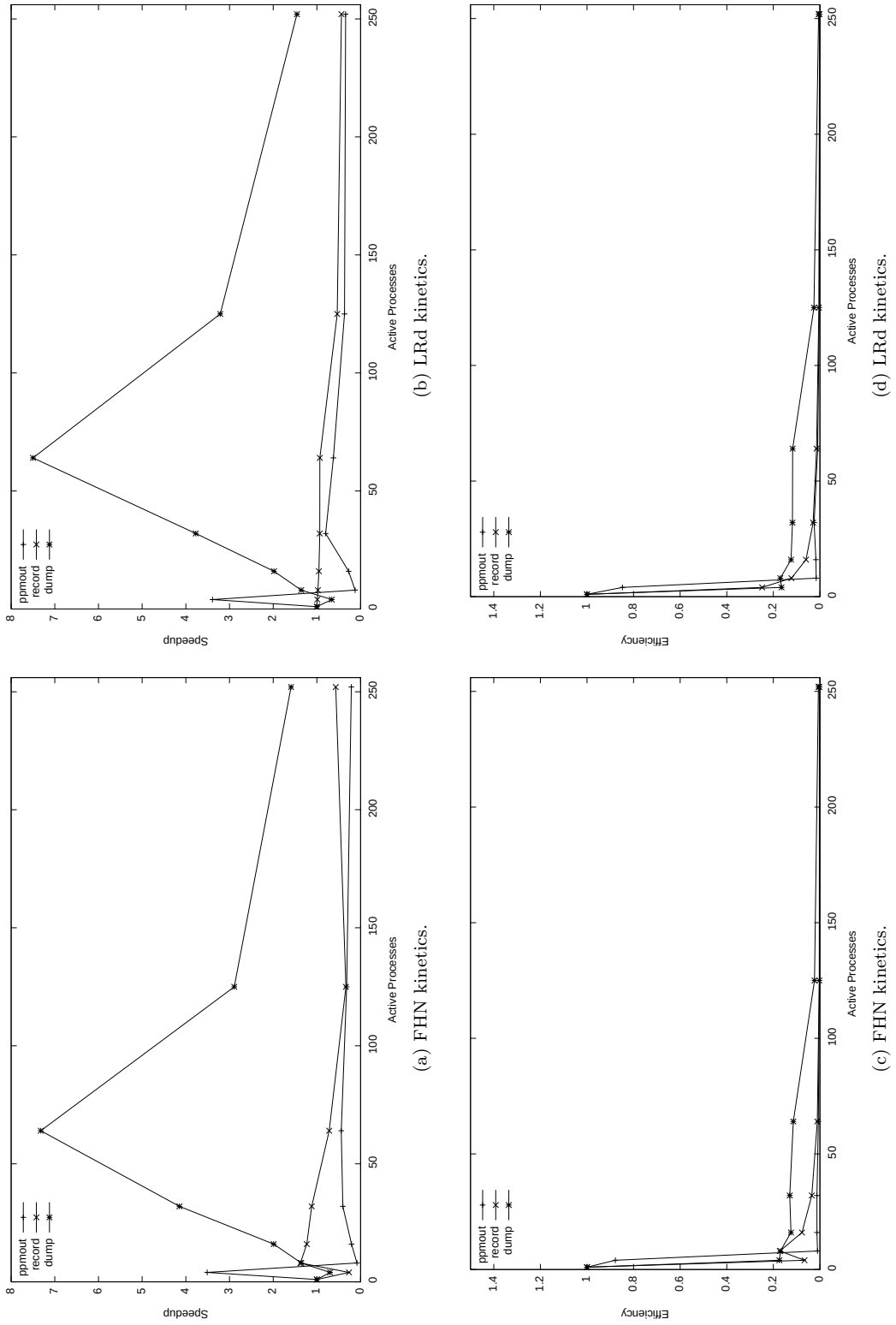


(a) Speedup



(b) Efficiency

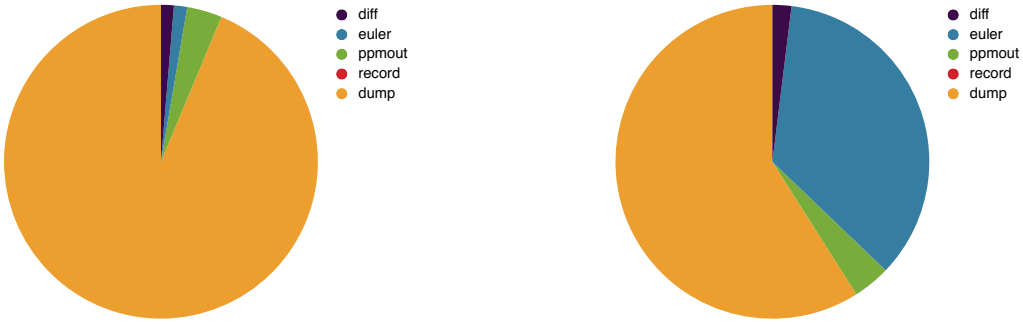
Figure 3.20: Comparison of `diff` performance using FHN and LRd kinetics.



(a) FHN kinetics. (b) LRd kinetics.

(c) FHN kinetics. (d) LRd kinetics.

Figure 3.21: Scaling performance of I/O devices.



(a) FHN kinetics.

(b) LRd kinetics.

Figure 3.22: Share of runtime by device on 1 process.

It could be argued that the relative efficiency of Beatbox's computational methods shifts the balance of runtime towards I/O and thus hampers scaling. The vastly improved top-line speedup (Figure 3.17a) of simulations using a more complex RHS module shows that reducing the proportion of runtime devoted to I/O allows Beatbox to scale more effectively. For this reason, any future developments of Beatbox that increase its computational complexity, e.g. the addition of bidomain tissue models, should also improve its scaling performance.

At present, Beatbox is best suited to simulations involving computationally complex RHS models on large meshes. Output should be as infrequent as possible, preferably using `dump`.

3.16 Extending Beatbox

Beatbox inherits QUI's device and RHS APIs, allowing third-party developers to add new functionality in the form of additional devices or RHS modules. As RHS modules operate on individual points of the mesh, their operation is unaffected from QUI. Devices, on the other hand, must take care to avoid any operations that could compromise results when run in parallel. Beatbox's usability and portability relies upon devices that operate correctly and identically in both sequential and parallel modes. This section discusses the rules to which developers should adhere when coding a new Beatbox device.

As was the convention in QUI, a device should only operate on its space, as specified by the user. This ensures that the effects of the device are as expected for the user. In Beatbox, this condition comes with the added incentive that a device that operates on points outside its space may find that the point is not present in the local subdomain. Changing values at points outside of the device's space may also lead to changed values at the interior halo points. As these points are expected to be read-only, they are not synchronised, and the new value will not be available to neighbouring processes. For the same reasons, a device may reference points immediately outside its space, with a shift of no more than one on any axis, but any points referenced in this fashion must be treated as read-only. A device that references neighbouring points will require that its halos are up-to-date, and so must include `DEVICE_REQUIRES_SYNC` in its Run function.

As discussed in Section 3.6.1, it is imperative that k_variables and other global values remain consistent on all processes. For example, an alteration to `t` on one process could cause deadlock problems if some processes arrive at a collective communication call while others 'skip' it. For this reason, a device must not alter any values other than those in permitted regions of `New` and its parameter structure without explicit parallelisation. Similarly, files on disk behave like global values. The complications of reading and writing to file are greatly complicated by parallelisation.

Unless explicitly parallelised, devices should treat all files as read-only.

Devices must also adhere to the conditions for parallel safety discussed in Section 3.6.1. Without explicit parallelisation, devices must not assign values to their parameter structure that could vary in each subdomain, i.e. no local-to-global assignment. Also, all operations on `New` must be commutative.

Finally, as a matter of pragmatism, messages to be printed to the standard output should be issued using the `MESSAGE()` function macro, to ensure that the same message is not repeated by all active processes.

The parallel-safe 'laws of devices' are listed more succinctly below:

1. A device must only alter layers of `New` in its space, at points in its space.
2. All operations on `New` must be commutative — the order of operation cannot be assumed.
3. Aside from the permitted regions of `New`, a device may only alter the values of local variables or those in its parameter structure.
4. Assignments made to a device's parameter structure must not be derived from data in `New`.
5. Assignments made to a device's parameter structure must not be derived from data in its `Space` structure.
6. Assignments made to a device's parameter structure must not be derived from local minima or maxima (e.g. `local_xmax`).
7. Assignments made to a device's parameter structure must not be derived from a random number generator.
8. From any point in its space, a device may not reference points in `New` beyond $\{x \pm 1, y \pm 1, z \pm 1\}$.
9. A device that references neighbouring points in `New` must put the `DEVICE_REQUIRES_SYNC` macro in its Create function.
10. Files accessed from a device must only be read.

Developer documentation for Beatbox is given in Appendix I.

3.17 Summary

In this chapter, we have introduced Beatbox, an extension of QUI, parallelised for distributed memory machines. We have examined the processes by which Beatbox decomposes the problem domain, and adjusts the function of its devices accordingly.

We have studied the need for, and mechanisms of inter-process communication using halo exchanges. To avoid unnecessary exchanges, we introduced the 'Magic Corners' algorithm, to exchange the corner points of halos.

The `k_func` device was studied in particular detail, to ensure that user-defined functions will yield identical results when run sequentially and in parallel. The necessary safety properties for `k_func` programs were defined and the consequent refinements to code parsing discussed. We introduced the `sample` and `reduce` devices to support functions unsafe for use in `k_func`. The result is that the same Beatbox script can be used to run the simulation sequentially or in parallel, without modification, and either method will produce identical results.

We discussed the modification of input/output devices for use in parallel and described the method of file partitioning used.

We saw that Beatbox's centralised model of domain decomposition allows some devices to benefit from *implicit parallelism*, i.e. they are parallelised with little or no modification to their code. For devices that are not explicitly parallelised, we revised the 'laws of devices' to maintain the safety of operations in parallel. To simplify device development, Beatbox introduces a set of literate macros, allowing the developer to signal the intended use of the device in a way that avoids redundant code and reduces the possibility of errors.

We discussed the validation of Beatbox's parallel implementation and examined is parallel performance. We saw that core computation scaled admirably. We also saw that scaling was limited by Input/Output and made suggestions for its future improvement.

Beatbox represents a significant step forward, enabling QUI's approachable scripting language and modular design to be used with High Performance Computing facilities. In the next chapter, we extend Beatbox's feature set further, allowing its simulations to take shape.

CHAPTER

FOUR

ANATOMICALLY REALISTIC TISSUE GEOMETRY

4.1 Introduction

Beatbox aims to provide users with the tools to run simulations on anatomically realistic meshes, including the inhomogeneities introduced by tissue structure. Modelling anatomically realistic geometry includes considerations of both the tissue's superficial shape and its underlying structure, specifically the directions of the muscle fibres that make up the tissue wall. It should be noted, however, that this chapter will not consider the modelling of microscopic tissue heterogeneities.

Numerical simulations in anatomically realistic meshes require two modifications to their implementations: methods to satisfy Neumann boundary conditions must take account of the irregular medium and its smooth, continuous boundary; and simulating the anisotropic myocardium requires modifications to the Laplace operator. This chapter discusses the aims of the anatomically realistic geometry features of Beatbox, the techniques used to implement them, and ultimately evaluates their success.

4.2 Reaction-Diffusion Equations & Neumann Boundary Conditions

As discussed in Section 1.7.2, our reaction-diffusion system is described by

$$\frac{\partial \mathbf{u}}{\partial t} = f(\mathbf{u}) + \nabla \cdot \mathbf{D} \nabla \mathbf{u} \tag{4.1}$$

on a finite domain, with no flux boundary conditions

$$\mathbf{n} \cdot \mathbf{D} \nabla \mathbf{u} \Big|_G = 0. \tag{4.2}$$

where G is the boundary, $\mathbf{n} = \{\alpha, \beta, \gamma\}$ is the vector normal to the boundary, and $\mathbf{D}$ is the diffusion tensor. $\mathbf{D}$ is a 3×3 symmetrical matrix.

4.2.1 Isotropic Diffusion

In the isotropic case, the diffusion matrix is diagonal,

$$\mathbf{D} = \begin{pmatrix} D & 0 & 0 \\ 0 & D & 0 \\ 0 & 0 & D \end{pmatrix},$$

allowing $\mathbf{D}$ to be treated as a scalar, such that

$$\nabla \cdot \mathbf{D}\nabla\mathbf{u} = D\nabla^2\mathbf{u} = D\left[\frac{\partial^2\mathbf{u}}{\partial x^2} + \frac{\partial^2\mathbf{u}}{\partial y^2} + \frac{\partial^2\mathbf{u}}{\partial z^2}\right] \tag{4.3}$$

with the Neumann boundary condition in the form of

$$\mathbf{n} \cdot \nabla\mathbf{u}\Big|_G = 0. \tag{4.4}$$

In a cuboid mesh, the domain can be defined precisely as a set of uniform constraints on the coordinate axes. Consequently, $\mathbf{n}$ will always lie parallel to one coordinate axis. Anatomically realistic meshes, however, present irregular boundaries, in which $\mathbf{n}$ will vary continuously. We can view components of $\mathbf{n}$ as weights of the derivatives along their corresponding axes:

$$\frac{\partial\mathbf{u}}{\partial\mathbf{n}} = \frac{\partial\mathbf{u}}{\partial x}\alpha + \frac{\partial\mathbf{u}}{\partial y}\beta + \frac{\partial\mathbf{u}}{\partial z}\gamma = 0. \tag{4.5}$$

The implementation of boundary conditions on the anisotropic medium are discussed in Section 4.9.

4.2.2 Anisotropic Diffusion

We shall consider axially symmetrical diffusion along fibres of the tissue. In the system of coordinates local to the fibre, the diffusion tensor is diagonal:

$$\hat{\mathbf{D}} = \begin{pmatrix} D_{\parallel} & 0 & 0 \\ 0 & D_{\perp} & 0 \\ 0 & 0 & D_{\perp} \end{pmatrix}$$

where $D_{\parallel}$ and $D_{\perp}$ are diffusion coefficients along and orthogonal to the fibre, respectively.

To obtain the diffusion tensor $\mathbf{D}$ in the laboratory system of reference $\{x, y, z\}$ we use transformation of coordinates.

$$\mathbf{DS} = \mathbf{S}\hat{\mathbf{D}} \tag{4.6}$$

where

$$\mathbf{S} = \begin{pmatrix} \alpha & b_1 & g_1 \\ a_2 & b_2 & g_2 \\ a_3 & b_3 & g_3 \end{pmatrix},$$

and the frame of reference characterised by three basis vectors: the fibre direction unit vector, $\mathbf{A} = (a_1, a_2, a_3)^T$, and two arbitrary unit vectors, $\mathbf{B} = (b_1, b_2, b_3)^T$ and $\mathbf{G} = (g_1, g_2, g_3)^T$, in the plane perpendicular to the fibre and orthogonal to each-other.

As $\mathbf{S}$ is orthogonal, $\mathbf{S}^{-1} = \mathbf{S}^T$. We can multiply (4.6) from the right by $\mathbf{S}^T$

$$\mathbf{DSS}^T = \mathbf{S}\hat{\mathbf{D}}\mathbf{S}^T \tag{4.7}$$

$$\therefore \mathbf{D} = \mathbf{S}\hat{\mathbf{D}}\mathbf{S}^T \tag{4.8}$$

For convenience, we also represent $\hat{\mathbf{D}}$ as the sum of two matrices,

$$\hat{\mathbf{D}} = \begin{pmatrix} D_\parallel & 0 & 0 \\ 0 & D_\perp & 0 \\ 0 & 0 & D_\perp \end{pmatrix} \tag{4.9}$$

$$= \begin{pmatrix} D_\parallel - D_\perp & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + D_\perp \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{4.10}$$

thus

$$\mathbf{D} = \mathbf{S}\hat{\mathbf{D}}\mathbf{S}^T$$

$$= \mathbf{S}\left[\begin{pmatrix} D_\parallel - D_\perp & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + D_\perp \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\right]\mathbf{S}^T$$

$$= \begin{pmatrix} a_1 & b_1 & g_1 \\ a_2 & b_2 & g_2 \\ a_3 & b_3 & g_3 \end{pmatrix}\left[\begin{pmatrix} D_\parallel - D_\perp & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + D_\perp \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\right]\mathbf{S}^T$$

$$= \left[\begin{pmatrix} a_1(D_\parallel - D_\perp) & 0 & 0 \\ a_2(D_\parallel - D_\perp) & 0 & 0 \\ a_3(D_\parallel - D_\perp) & 0 & 0 \end{pmatrix} + D_\perp\mathbf{S}\right]\mathbf{S}^T$$

$$= \begin{pmatrix} a_1(D_\parallel - D_\perp) & 0 & 0 \\ a_2(D_\parallel - D_\perp) & 0 & 0 \\ a_3(D_\parallel - D_\perp) & 0 & 0 \end{pmatrix}\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ g_1 & g_2 & g_3 \end{pmatrix} + D_\perp\mathbf{I}$$

$$= \begin{pmatrix} a_1^2(D_\parallel - D_\perp) & a_1a_2(D_\parallel - D_\perp) & a_1a_3(D_\parallel - D_\perp) \\ a_1a_2(D_\parallel - D_\perp) & a_2^2(D_\parallel - D_\perp) & a_2a_3(D_\parallel - D_\perp) \\ a_1a_3(D_\parallel - D_\perp) & a_2a_3(D_\parallel - D_\perp) & a_3^2(D_\parallel - D_\perp) \end{pmatrix} + D_\perp\mathbf{I}$$

$$= (D_\parallel - D_\perp)\begin{pmatrix} a_1^2 & a_1a_2 & a_1a_3 \\ a_1a_2 & a_2^2 & a_2a_3 \\ a_1a_3 & a_2a_3 & a_3^2 \end{pmatrix} + D_\perp\mathbf{I},$$

where $\mathbf{I}$ is the identity matrix $\left(\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{smallmatrix}\right)$.
Finally, as in Clayton and Panfilov [2008],

$$\mathbf{D} = D_\perp\mathbf{I} + (D_\parallel - D_\perp)\mathbf{A}\mathbf{A}^T, \tag{4.11}$$

The implementation of anisotropic diffusion and associated boundary conditions in Beatbox is discussed in Section 4.10.

4.3 Software Implementation Overview

The software representation of anatomically realistic tissue geometry will require two sets of additional data to be stored and queried: an incidence matrix, indicating whether points of the mesh are in or out off the tissue; and, as $\mathbf{A} = f(x, y, z)$, a vector describing local fibre orientation for each point.

4.3.1 Aims of the Software Implementation

In addition to more accurately modelling cardiac tissue, the implementation of realistic geometry described below has the following aims;

1. Realistic tissue geometry should be enabled and disabled with only small changes to the user script
2. Meshes should be easily interchangeable
3. Meshes should be simple to describe
4. The system should be accommodating to meshes with minor imperfections
5. The implementation should present a simple, intuitive API to third-party developers

Definitions

Throughout this chapter, references will be made to three classifications of points in a mesh, illustrated in Figure 4.1:

Void points represent free space surrounding the heart, or inside its chambers. No values are computed for void points, which are considered to be absent.

Tissue points are those points that form the tissue. Any point that is considered to be part of the tissue, whether intramural or on the wall surface, is classified as tissue.

Irregular points are a subset of tissue points, who have one or more void neighbours. The neighbourhood under consideration will depend on the scheme in question, and is discussed below.

Tissue points that are not irregular are referred to as internal tissue points.

4.4 User Interface

As described above, representing realistic geometry requires the annotation of a mesh to include details of tissue presence and fibre or boundary normal directions at each point in the tissue. Ordinarily, `New` is used to store any data that occur per point. However, as `vmax`, the number of variable layers for each point in `New` is user-defined, the user would be required to understand and account for the allocation of geometry data in their script. Such a requirement would appear to contradict Aim 1. In order to maintain simple switching between parallelepipedal slabs and anatomically realistic tissue, geometry data are kept separate from `New` and does not affect the value of `vmax`.

To satisfy Aim 2, meshes are described in separate geometry files and loaded at runtime. The format of a geometry file is described in Section 4.6. Realistic geometry is activated in the

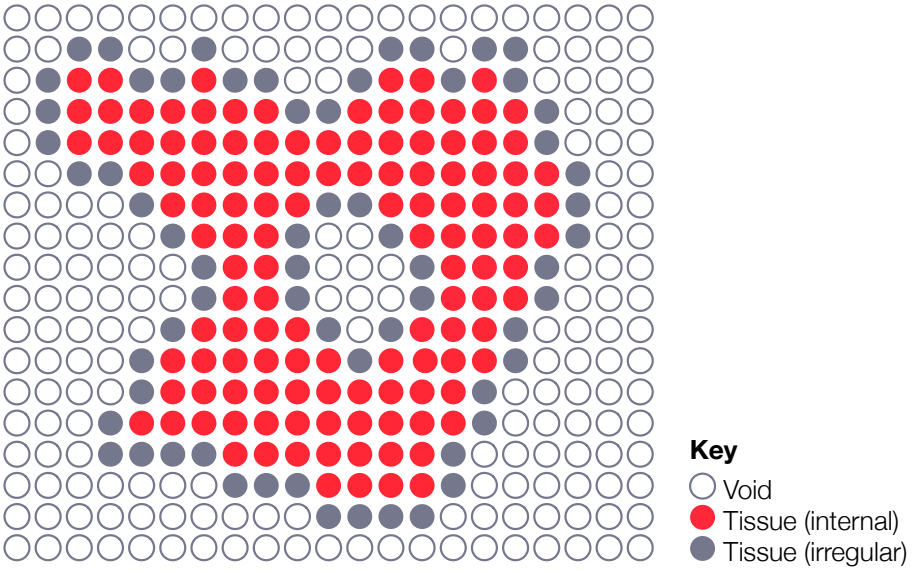


Figure 4.1: Points in an irregular mesh.

user script by specifying the geometry file to be used to the `geometry` parameter of the `state` module, as shown in Listing 4.1. If the `geometry` parameter is supplied, `xmax`, `ymax` and `zmax` are determined by the data in the geometry file. If the user attempts to supply values for `xmax`, `ymax` or `zmax` in addition to `geometry`, the user is presented with a message warning them that specified values for `xmax`, `ymax` and `zmax` will be ignored. The user will specify `vmax` as normal.

```
state geometry=rabbit.bbg vmax=3;
```

Listing 4.1: The `state` module being called with geometry file *rabbit.bbg*.

4.5 Data Structures

Data describing the realistic geometry is held in memory throughout the simulation run. For each point in the mesh, an integer status variable indicates if the point is in or out of tissue, and three double-precision floating point variables hold components of the local fibre vector. The status of points and their corresponding vectors are stored separately from `New` in a similar four-dimensional array named `Geom`. While `New` is allocated `xlen*ylen*zlen*vmax` values, `Geom` is allocated `xlen*ylen*zlen*geom_vmax` variables, where `geom_vmax` is `4` (for the point's status and its three vector components). Macros, listed in Table 4.1, define their ordering and the meaning of status values.

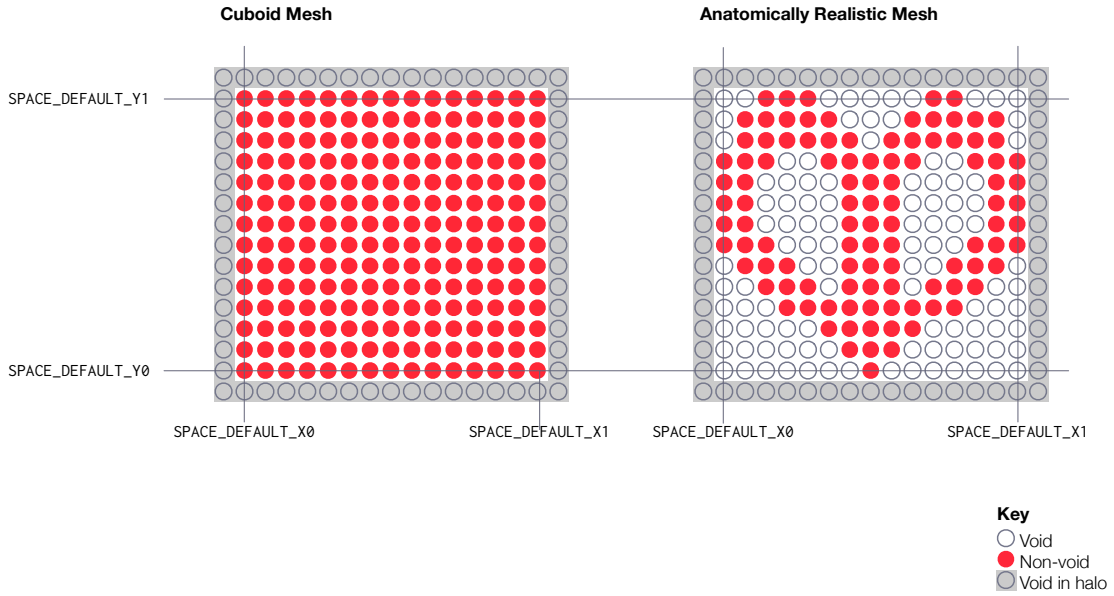


Figure 4.2: Effect of coordinates on tissue status in cuboid and anatomically realistic meshes.

Name	Value
Layers of `Geom`	
`GEOM_STATUS`	0
`GEOM_FIBRE_1`	1
`GEOM_FIBRE_2`	2
`GEOM_FIBRE_3`	3
Point Statuses	
`GEOM_VOID`	0
`GEOM_TISSUE`	1

Table 4.1: Constants for geometry data.

`Geom` is accessed in much the same way as `New`, using an index macro, `geom_ind()` (*state.h*). Pseudocode for `geom_ind()` is given in GeomInd below:

GeomInd(x, y, z, v)

1 **return** $((x + \text{ONE}) - local\_xmin) \times \text{GEOM\_VMAX\_ZMAX\_YMAX} +$
$((y + \text{TWO}) - local\_ymin \times \text{GEOM\_VMAX\_ZMAX} +$
$((z + \text{TRI}) - local\_zmin) \times \text{GEOM\_VMAX} + v$

Since the default device space omits points at the absolute extremes of the medium, the geometry implementation maintains consistent behaviour by ensuring that all points at the extremes of the medium are void. If necessary, this will be achieved by 'wrapping' the tissue points in a single layer of void points, as shown in Figure 4.3 and discussed below.

The macro function `isTissue()` is defined to test the status of a point given its coordinates. As illustrated by Figure 4.2, this function provides equivalent behaviour in cuboid meshes by

testing the point's coordinates against the default device space. If the point falls outside of this space, the function returns `0`. Note that `Geom` does not hold any record of irregular points; all tissue points, either internal or irregular, have statuses of `GEOM_TISSUE` and supplying their coordinates to `isTissue()` will return `1`. Pseudocode for the `isTissue()` macro is given in IsTissue, below, and its code can be seen in Listing E.2.

IsTissue(x, y, z)
1 **if** $x \geq$ X0 and $x \leq$ X1 and
$y \geq$ Y0 and $y \leq$ Y1 and
$z \geq$ Z0 and $z \leq$ Z1 and
(GeometryOn $\neq 1$ or $Geom$[GeomInd(x, y, z, Status)] = Tissue)
2 **then**
3 **return** True
4 **else**
5 **return** False

4.5.1 `Geom` and MPI

Although `Geom` has many similarities with `New`, it should be noted that the data in `Geom` is not liable to change over the course of a simulation run. For this reason, there is no need to synchronise geometry data between subdomains when using MPI.

There will, however, be some need to test the geometry status of neighbouring points, which will inevitably lead to tests being made on points immediately outside of the simulation medium. For this reason, `Geom` is allocated the same number of points as `New`, including halo points, to allow geometry data from neighbouring points to be referred to.

4.6 Geometry Files

By representing a geometric mesh in a file, users may exchange meshes as easily as they do simulation scripts. The file format used has been designed to reduce redundancy, and thus any conflict between redundant values. For example, rather than require users or geometry file authors to specify the dimensions of the mesh, the dimensions are implied from the coordinates of points in the geometry file.

Geometry files — which by convention use the *.bbg* (Beatbox geometry) extension — contain comma-separated values, with each record being separated by a newline. Lines are of the form:

```
x,y,z,status,fibre1,fibre2,fibre3
```

where `status` is an integer with value equivalent to `GEOM_VOID` or `GEOM_TISSUE` (*state.h*) and `fibre*` is a component of the vector defining the local fibre direction.

Where points lie inside the tissue, their status value should be equal to `GEOM_TISSUE`. The vector components should describe a unit vector. In tissue points, the vector gives the direction of the tissue fibre at that location, while void points' vectors will be ignored. To limit the size of the files, points not specified will be assumed to be void, such that only tissue points need to be specified. Data for each point should be given only once. Handling of invalid geometry files is described in Section 4.6.1

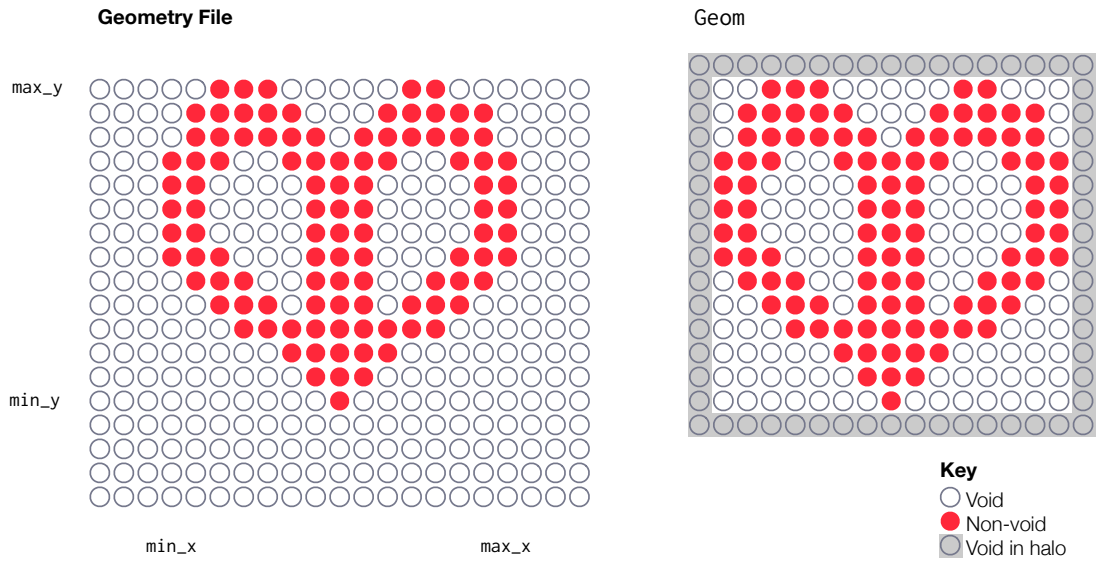


Figure 4.3: Adjusting the simulation medium to fit the tissue. The image on the left shows a mesh as described by the geometry file. On the right is the same mesh, as it would be represented in `Geom`.

4.6.1 Reading Geometry Files

A geometry file is read in two stages: firstly, the mesh is measured to define the dimensions of `New` and `Geom`, then `Geom` is populated with data from the file. The functions involved are `getMeshDimensions()` and `populateMesh()` respectively, both of which are defined in *geometry.c*.

`getMeshDimensions()` assigns values to `xmax`,`ymax` and `zmax` according to the dimensions of the mesh, with the aim that the tissue should be surrounded on all sides by a single layer of void points. Where the number of void points at the extremes of the mesh is not equal to 1, the mesh will be adjusted accordingly, as shown in Figure 4.3. The function begins by reading the entire geometry file to find the minimum and maximum x, y and z coordinate of tissue points. These values are then used to establish the dimensions of the medium. To facilitate cropping, the minimum coordinate values in each axis are used to compute `x_offset`, `y_offset` and `z_offset`, which in turn are used by `populateMesh()` to adjust incoming coordinates such that the first tissue point on the x, y and z axes will have coordinates `ONE`, `TWO` and `TRI` respectively. Consequently, all points in a 2D mesh will have z-coordinates of `0`.

In `populateMesh()` (*geometry.c*), `Geom` is first initialised by setting every point in the simulation medium to be void. The geometry file is then read line by line. In MPI, each process reads the file in full, only assigning values to `Geom` when the point being read falls within the local subdomain or its halos.

Handling and Correcting Errors

As discussed above, the design of the geometry file structure aims to reduce the opportunity for error. There are three types of error, however, that may still arise: invalid status values; invalid fibre vectors; and duplicate points.

Invalid status values are treated as unrecoverable errors. Rather than attempt to guess the

geometry file author's intention, if Beatbox encounters an invalid status value for a point, reading stops immediately and the user is alerted.

Invalid fibre vectors are handled more flexibly. If the fibre vector provided is not a unit vector, it is likely that the user would still like the fibre at that point to indicate the same direction. To facilitate this, the `normaliseVector()` function in *geometry.c* can be used to produce a unit vector with the same direction as the vector from the file. If the given vector has a length of 0, an error is raised that will cause Beatbox to identify the problem point to the user and quit. To better inform the user of the correctness of their geometry file, automatic normalisation of fibre vectors will only take place if the `normaliseVectors` parameter of the `state` module is set to `1` in the script.

If status and fibre values are given for the same point multiple times, the final set of values will be used. This will happen silently, without warning the user.

4.7 Adapting `euler` for Anatomically Realistic Simulations

The majority of Beatbox devices can operate correctly on an anatomically realistic simulation medium without modification. However, as anatomically realistic meshes contain a large proportion of void points, the efficiency of a device can be improved by making it operate only on those points that will contribute to the solution. This is especially true of the `euler` device, as RHS computations are likely[1] to be responsible for a large proportion of a simulation's runtime.

Using the `isTissue()` macro, the `euler` device is modified to compute reaction on only tissue points. The `euler` device's run function iterates as normal over points in `New`, however, the RHS function is only called for a point if `isTissue()` returns true for the point's coordinates.

4.8 Adapting `ppmout` for Anatomically Realistic Simulations

The image files produced by the `ppmout` device display every point in the mesh. For visualisation purposes, it will be useful to identify the pixels of the image that relate to tissue points. To facilitate this, the `ppmout` device is adapted for geometry by allowing the user to specify a background colour to be assigned to pixels that correspond to void points.

As shown in Listing 4.2, the user specifies the red, green and blue values of the background colour using the `bgr`, `bgg`, `bgb` parameters respectively. Colour values at tissue points are computed in the same manner as for cuboid meshes.

```
ppmout when=out file="ppm/%04d.ppm" mode="w"
    r=[u] r0=umin r1=umax
    g=[v] g0=vmin g1=vmax
    b=[i] b0=0 b1=255
    bgr=0 bgg=0 bgb=255
;
```

Listing 4.2: A `ppmout` device call with a blue background colour.

[1] The contribution of RHS computations to a simulation's runtime will depend on the RHS module in use, the other devices that form the simulation, and the performance of the machine on which the simulation is run.

4.9 Neumann Boundary Conditions in the Isotropic Medium

In a cuboid mesh, the boundary normal $\mathbf{n}$ is always parallel to one coordinate axis. For example, points at the upper x boundary will have $\mathbf{n} = \{1, 0, 0\}$, while points at the lower y boundary will have $\mathbf{n} = \{0, -1, 0\}$. This affords us the opportunity to simplify the means by which we achieve no-flux boundary conditions. In realistic tissue geometry, however, tissue surfaces are irregular and boundary normals vary continuously.

The method proposed by Buist et al. [2003] replaces two point central difference with a three point one-sided at irregular points, where one point is missing in the direction of interest, i.e. at boundaries. Some finite difference meshes can be unsuited to such a scheme, as some non-void points may be straddled by void points in one or two directions, preventing suitable interior neighbours from being found. The rabbit mesh under discussion in this chapter has several such points. To prevent unnecessary restriction on the meshes suitable for use in Beatbox, an alternative method was sought.

Two candidate solutions for boundary conditions in irregular meshes were implemented and evaluated in Beatbox. The following sections discuss the algorithms involved and their relative merits for the intended application. The methods presented below differ in that the first computes a boundary for irregular points explicitly, while the second, preferred, method modifies the Laplacian algorithm at irregular points to satisfy the Neumann boundary condition implicitly.

4.9.1 Explicit Method

Values of $\mathbf{u}$ can be deduced from (4.5) at irregular points. For a given irregular point, the value is computed as the weighted sum of neighbouring points. In a cuboid medium, this would equate to points at the end of an axis being assigned values from the penultimate points. For realistic geometry, we can use the components of $\mathbf{n}$, $\{\alpha, \beta, \gamma\}$, to interpolate the partial derivatives using values from neighbouring points. So (4.5) can be discretised to

$$\begin{aligned}\frac{\partial \mathbf{u}}{\partial \mathbf{n}} = {} & \alpha[\mathrm{sgn}(\alpha)\mathbf{u}_{\ldots} - \mathrm{sgn}(\alpha)\mathbf{u}_{(-\mathrm{sgn}(\alpha))\ldots}]+ \\ & \beta[\mathrm{sgn}(\beta)\mathbf{u}_{\ldots} - \mathrm{sgn}(\beta)\mathbf{u}_{\cdot(-\mathrm{sgn}(\beta))\cdot}]+ \\ & \gamma[\mathrm{sgn}(\gamma)\mathbf{u}_{\ldots} - \mathrm{sgn}(\gamma)\mathbf{u}_{\ldots(-\mathrm{sgn}(\gamma))}] \qquad (4.12) \\ & = |\alpha|\ \mathbf{u}_{\ldots} - \mathbf{u}_{-\mathrm{sgn}(\alpha)\ldots}+ \\ & |\beta|\ \mathbf{u}_{\ldots} - \mathbf{u}_{\cdot-\mathrm{sgn}(\beta)\cdot}+ \\ & |\gamma|\ \mathbf{u}_{\ldots} - \mathbf{u}_{\ldots-\mathrm{sgn}(\gamma)} \qquad (4.13) \\ & = 0 \qquad (4.14)\end{aligned}$$

as $\mathrm{sgn}(i)i = |i|$.

Therefore, from (4.12–4.14),

$$\therefore \mathbf{u}_{\ldots} = \frac{|\alpha|\mathbf{u}_{-\mathrm{sgn}(\alpha)\ldots} + |\beta|\mathbf{u}_{\cdot-\mathrm{sgn}(\beta)\cdot} + |\gamma|\mathbf{u}_{\ldots-\mathrm{sgn}(\gamma)}}{|\alpha| + |\beta| + |\gamma|} \qquad (4.15)$$

For example, $\mathbf{u}_{-sign(\alpha)\ldots}$, given a negative value for α, is the value of $\mathbf{u}$ at position $x + 1$. Intuitively, (4.15) gives that the boundary value will be computed as the weighted mean of neighbouring points in the opposite direction along each axis to that given by the boundary normal, as shown in Figure 4.5. The application of this method in Beatbox requires two additional sets of information to be acquired: the identity of irregular points and $\mathbf{n}$ at each of those points.

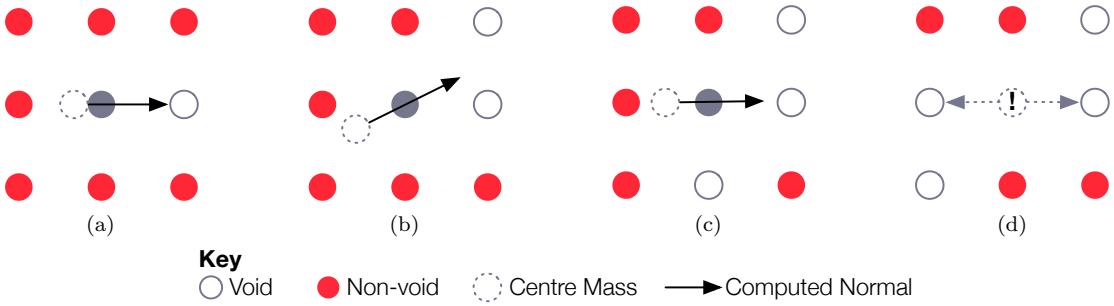


Figure 4.4: Centre mass and resultant boundary normals.

Identifying Irregular Points

In order to avoid references to void points in Laplacian computations, we identify irregular points in the mesh. Irregular points are those with one or more void neighbour. The neighbourhood to be considered depends on the stencil of the Laplacian algorithm used, as it is the Laplacian algorithm that will make references to neighbouring points. As Beatbox's `diff` device employs a six-point 'cross' stencil, we need only test the status of these points when identifying irregular points.

Computing Boundary Normals

Relatively few of the available models of cardiac geometry include boundary normals. For this reason, boundary normals must usually be estimated from the geometry data, particularly the statuses of points. The technique used by Beatbox aims to find the 'centre mass' of the irregular point's neighbourhood — i.e. the average coordinates of the irregular point's tissue neighbours. The boundary normal can then be drawn from this location, through the irregular point and out of the tissue. The neighbourhood used for normal generation need not match the Laplacian stencil. To make the most of the available data, the implementation of this method in Beatbox uses all 26 immediate neighbours.

One potential pitfall of the 'centre mass' method is that there can arise situations in the data where, despite the presence of void points, the arrangement of tissue points around the irregular point is such that their average location will coincide with the irregular point, as illustrated in Figure 4.4. This prevents a meaningful boundary normal from being generated. In these cases, the boundary normal can be made to point directly towards any of the void points.

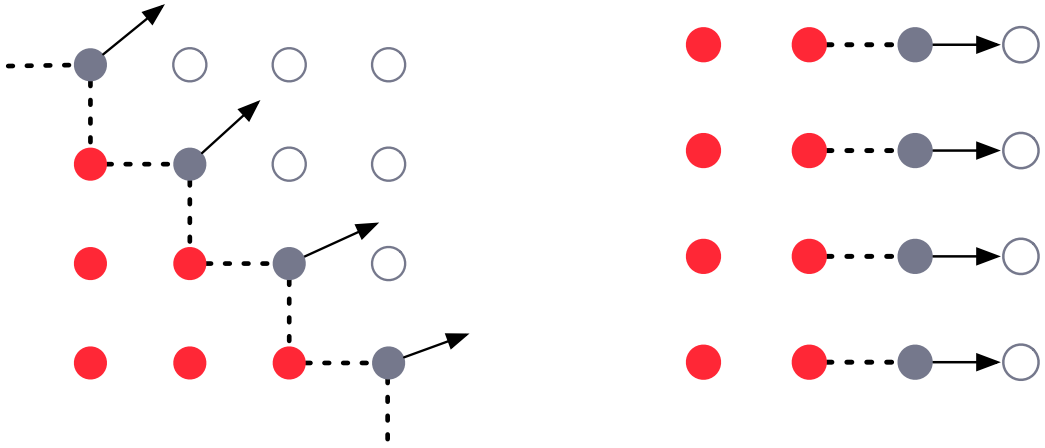
The values of boundary normals give a finer-grained description of the curvature of the medium than can be represented by the rasterised mesh. These differing resolutions present some difficulties in the computation of boundary values at some locations, where one or two of the desired neighbouring points are void.

Selecting Neighbours

For each neighbouring point to be found, a test is made at the desired location of that neighbour to see if the point is tissue. If so, the point can be used as a neighbour. If the point is void, a hierarchy of alternatives are checked until a tissue point is found.

For $\mathbf{u}_{-sign(\alpha)..}$, where α, β and γ are positive, and $\alpha > \beta > \gamma$, the hierarchy is detailed below. Each point in the hierarchy is tested, in sequence, until a suitable neighbour is found.

- *Ideal neighbour.* Test the neighbour at $\{x-1, y, z\}$.



(a) (b)

Figure 4.5: Relationship between boundary normals and selected neighbours.

- *Select the diagonal, by adding a shift along the axis with the greater corresponding vector component.* Test the neighbour at $\{x-1, y-1, z\}$.
- *Select the diagonal, by adding a shift along the remaining axis with the lesser corresponding vector component.* Test the neighbour at $\{x-1, y, z-1\}$.
- *Shift along all three axes.* Test the neighbour at $\{x-1, y-1, z-1\}$.
- *Shift along the remaining axis with the greater corresponding vector component.* Test the neighbour at $\{x, y-1, z\}$.
- *Shift along the remaining axis with the lesser corresponding vector component.* Test the neighbour at $\{x, y, z-1\}$.
- *Change direction on the same axis.* Test the neighbour at $\{x+1, y, z\}$.

In the last case, where a neighbour is selected in the same direction as the boundary normal, its contribution to the boundary value equation is made negative. For example, if the x-neighbour were selected in this manner, (4.15) would be altered as shown in (4.16).

$$\mathbf{u}_{\cdots} = \frac{-|\alpha|\mathbf{u}_{-\mathrm{sgn}(\alpha)\cdot\cdot} + |\beta|\mathbf{u}_{\cdot-\mathrm{sgn}(\beta)\cdot} + |\gamma|\mathbf{u}_{\cdot\cdot-\mathrm{sgn}(\gamma)}}{-|\alpha| + |\beta| + |\gamma|} \tag{4.16}$$

Anisotropy

As described in Section 4.9, isotropy allows the simplification of the Neumann boundary condition to exclude the scalar diffusion coefficient. Since anisotropic diffusion employs a diffusion tensor, this must be considered when solving boundary conditions for anisotropic media, as shown in (4.17).

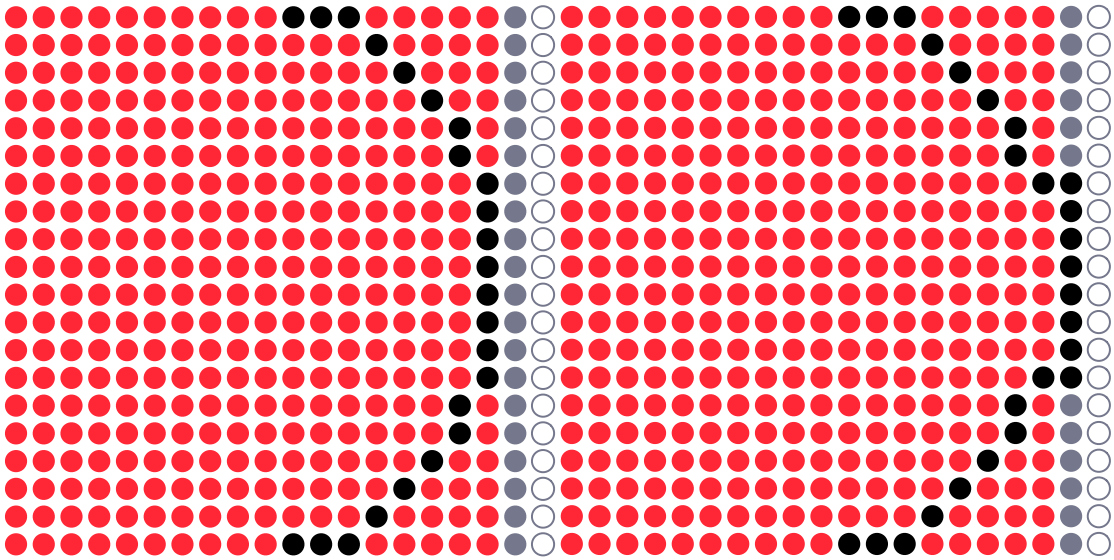
$$\mathbf{n} \cdot \mathbf{D}\nabla\mathbf{u}\Big|_G = 0, \tag{4.17}$$

For the reasons discussed below, the explicit method is not used to satisfy boundary conditions in Beatbox. Consequently, its implementation for anisotropy is not considered here.

Limitations of the Explicit Method

Computing boundary values explicitly for irregular points from their intramural neighbours can lead to some artefacts in the solution near the boundary. When a wave of excitation reaches the penultimate point on an axis, it will immediately 'leap' to an associated boundary point, along the boundary normal. This can distort the wavefront, creating fronts that travel directly along the boundary normal between the last points in the tissue.

There is a tradeoff to be made when selecting neighbouring points. A greater proportion of ideal neighbours are likely to be found if other irregular points are accepted as neighbours. This can, however, introduce additional error, as any error caused by the above issue can be allowed to spread to neighbouring irregular points.



(a) QUI `neum` device (b) Explicit method

Figure 4.6: Black points depict an excitation wavefront moving towards the boundary. In the explicit method, the wavefront 'leaps' from the last internal tissue point towards the irregular point.

4.9.2 Laplacian with Implicit Neumann Boundary Conditions

The second method under consideration involves computation of a modified Laplacian, taking void neighbours into account at irregular points.

Since flux between points is a result of the Laplace operator, we can modify its computation to prevent flux at boundaries. The discretisation of the Laplacian equation used by QUI's `diff3d` device is

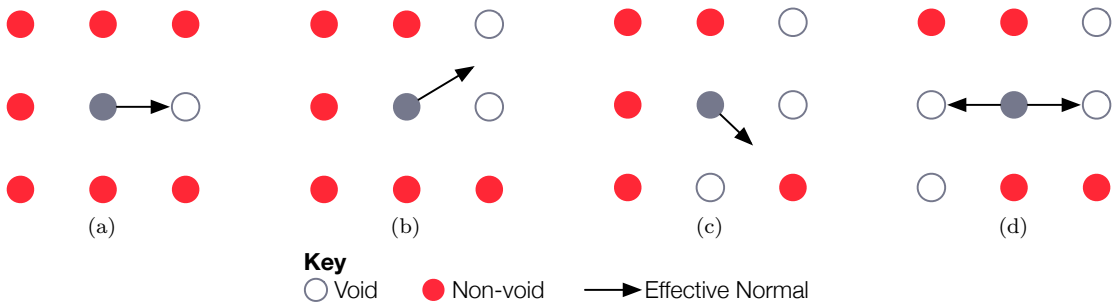


Figure 4.7: Effective boundary normals.

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} \tag{4.18}$$

$$= \left(\frac{\mathbf{D}}{\Delta^2}\right)(u_{+\cdot\cdot} + u_{-\cdot\cdot} + u_{\cdot+\cdot} + u_{\cdot-\cdot} + u_{\cdot\cdot+} + u_{\cdot\cdot-} - 6u_{\cdot\cdot\cdot}) \tag{4.19}$$

For points at the edges of the cuboid medium, one or more neighbours would refer to points updated by the `neum3d` device, which would have identical values to the centre point ($u_{\cdot\cdot\cdot}$), giving no flux at the boundary. To avoid orthogonality of conduction at the boundary, as described in Section 4.9.1, users may choose to exclude the points updated by `neum3d` from the output.

In realistic geometry, points immediately outside the tissue are likely to have more than one tissue neighbour. In these cases, points updated in this way would be insufficient to maintain zero flux. The technique presented here solves this problem by applying the Laplacian to all tissue points, with some modifications to incorporate irregular points.

$$\nabla^2 u = \frac{\mathbf{D}}{\Delta^2}(\delta^{+\cdot\cdot} + \delta^{-\cdot\cdot} + \delta^{\cdot+\cdot} + \delta^{\cdot-\cdot} + \delta^{\cdot\cdot+} + \delta^{\cdot\cdot-} - 6u_{\cdot\cdot\cdot}) \tag{4.20}$$

where δ returns the value from a neighbouring point, if present, or the centre value if not. Superindices indicate the relative position of the point being tested, e.g. for a centre point at {i,j,k}:

$$\delta^{\cdot+\cdot} = \begin{cases} u_{\cdot+\cdot} & \text{if } \{i, j+1, k\} \text{ is tissue} \\ u_{\cdot\cdot\cdot} & \text{if } \{i, j+1, k\} \text{ is void} \end{cases} \tag{4.21}$$

Effective Boundary Normals

Substituting the centre value for that of neighbouring void points ensures zero flux between the irregular point and *all* of its void neighbours. While no explicit boundary normal is used in this process, the result is an effective boundary normal pointing towards the average location of neighbouring void points. For example, given a single void neighbour in the positive x direction, the effective boundary normal would be $\{1, 0, 0\}$.

Limitations of Laplacian with Implicit Neumann Conditions

The effective boundary normals created by this method will be equivalent to those generated by the technique described in Section 4.9.1 using a six-point 'cross' stencil. By taking a broader

neighbourhood into account, other methods of generating boundary normals will be likely to yield a smoother, more biophysically realistic tissue surface.

4.9.3 Comparison of Boundary Conditions Methods

The methods described in Sections 4.9.1 and 4.9.2 above provide a practical implementation of boundary conditions on realistic geometry. The explicit method can be used with automatically generated boundary normals, or those provided from *in vivo* or *in vitro* measurements. However, the orthogonality of conduction at the boundary caused by this method may concern some users. It is possible that this issue could be overcome by introducing a diffusion coefficient to the boundary value computations. However, such a solution would require that an anisotropic diffusion tensor be computed at irregular points, which would greatly add to the complexity of the solution and its implementation.

While the modified Laplacian method can produce a coarse approximation of the tissue surface, its robustness with regard to the neighbourhood of points is welcome. It is also possible to see that the method can be more easily extended to anisotropy. Finally, there is a parallel performance consideration: given that the modified Laplacian method will obviate the need for a separate boundary conditions device, there is an opportunity to remove one complete halo swap from the average simulation timestep. This will reduce synchronisation time and benefit users on communication-bound machines.

For the above reasons, Beatbox implements the modified Laplacian method to handle irregular points in cuboid and geometric meshes. A full description of its implementation, and its extension to anisotropic diffusion are discussed below.

4.9.4 Implementation of Laplacian with Implicit Neumann Conditions

The aim in implementing this algorithm was to provide a general purpose device, capable of adapting to one-, two- and three-dimensional meshes, with or without realistic geometry. Doing so reduces the number of script calls that will have to be changed should the user wish to run their script with a different dimensionality. This generality is achieved by summing the values from neighbouring points, if present, on all active axes. If the points are not present, their value is substituted with that of the centre point. The sum of neighbouring values is then added to $-2du...$, where d is the dimensionality. Finally, the result is multiplied by D/Δ^2. Code for the function can be seen in Listing E.3.

4.10 Anisotropic Diffusion

The method described here computes the effect of fibre direction on conductance. It employs axial symmetry, meaning that the resultant conduction velocity is equal in all directions orthogonal to the fibre axis; the orientations of sheet planes are not considered.

4.10.1 Discretisation of Reaction Diffusion Equations for Anisotropic Diffusion

The Laplace operator must be modified to take account of local diffusion tensors. Clayton and Panfilov [2008] give

$$\nabla \cdot \mathbf{D}\nabla u = \sum_{i=1}^{3}\sum_{j=1}^{3} \frac{\partial}{\partial x_i}\left(d_{ij}\frac{\partial u}{\partial x_j}\right) = \sum_{i=1}^{3}\sum_{j=1}^{3}\left(\frac{\partial d_{ij}}{\partial x_i}\frac{\partial u}{\partial x_j} + d_{ij}\frac{\partial^2 u}{\partial x_i \partial x_j}\right), \tag{4.22}$$

for anisotropic diffusion in Cartesian coordinates $x_1,\ x_2,\ x_3$.

(4.22) can be solved by substituting the following approximations:

First derivatives:

$$\frac{\partial u}{\partial x} \approx \frac{u_{+..} - u_{-..}}{2\Delta} \tag{4.23}$$

$$\frac{\partial u}{\partial y} \approx \frac{u_{.+.} - u_{.-.}}{2\Delta} \tag{4.24}$$

$$\frac{\partial u}{\partial z} \approx \frac{u_{..+} - u_{..-}}{2\Delta} \tag{4.25}$$

Homogeneous second derivatives:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{+..} + u_{-..} - 2u_{...}}{\Delta^2} \tag{4.26}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u_{.+.} + u_{.-.} - 2u_{...}}{\Delta^2} \tag{4.27}$$

$$\frac{\partial^2 u}{\partial z^2} \approx \frac{u_{..+} + u_{..-} - 2u_{...}}{\Delta^2} \tag{4.28}$$

Inhomogeneous second derivatives:

$$\frac{\partial^2 u}{\partial x \partial y} \approx \frac{(u_{++.} - u_{+-.}) - (u_{-+.} - u_{--.})}{4\Delta^2} \tag{4.29}$$

$$\frac{\partial^2 u}{\partial x \partial z} \approx \frac{(u_{+.+} - u_{+.-}) - (u_{-.+} - u_{-.-})}{4\Delta^2} \tag{4.30}$$

$$\frac{\partial^2 u}{\partial y \partial x} \approx \frac{(u_{++.} - u_{-+.}) - (u_{+-.} - u_{--.})}{4\Delta^2} \tag{4.31}$$

$$\frac{\partial^2 u}{\partial y \partial z} \approx \frac{(u_{.++} - u_{.+-}) - (u_{.-+} - u_{.--})}{4\Delta^2} \tag{4.32}$$

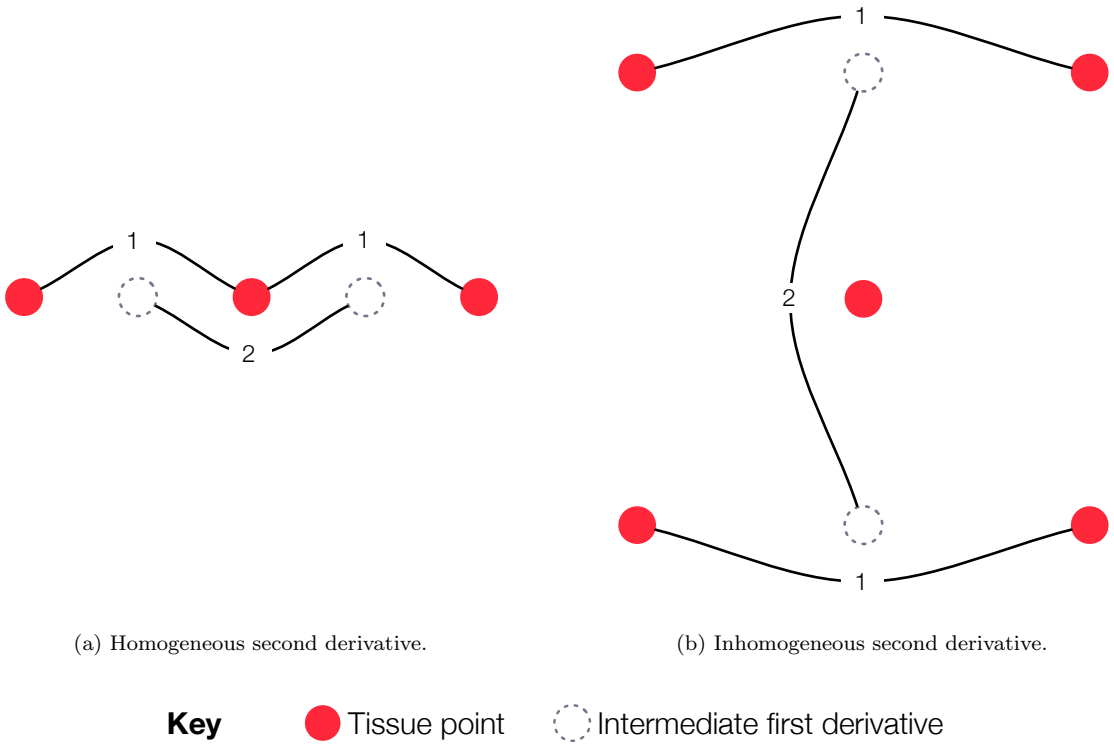
$$\frac{\partial^2 u}{\partial z \partial x} \approx \frac{(u_{+.+} - u_{-.+}) - (u_{+.-} - u_{-.-})}{4\Delta^2} \tag{4.33}$$

$$\frac{\partial^2 u}{\partial z \partial y} \approx \frac{(u_{.++} - u_{.-+}) - (u_{.+-} - u_{.--})}{4\Delta^2} \tag{4.34}$$

4.11 Anisotropic Diffusion with Implicit Neumann Boundary Conditions

At irregular points, some of the neighbouring points referred to in (4.23–4.34) will be void. The notion of using a delta function to isolate void points from diffusion can be applied to anisotropy

but, unlike the isotropic case, individual void points cannot be excluded individually. The use of inhomogeneous second derivatives as part of the anisotropic Laplacian computations, employs a 'capital I' shaped stencil, which will cause a diagonally neighbouring void point to be referenced several times, and in combination with other neighbouring points, as shown in Figure 4.8.



(a) Homogeneous second derivative.

(b) Inhomogeneous second derivative.

Figure 4.8: 'Cross' and 'Capital I' stencils for computation of second derivatives. Lines connect points involved in the computation of first and second derivatives.

After substituting (4.23–4.34) into (4.22), the terms can be regrouped to allow neighbours connected by a particular derivative to be excluded if one of more neighbours is void. This gives

$$
\begin{aligned}
&\frac{1}{4\Delta^2}\Big\{ \\
&\quad 4d^{11}_{\cdots}[u_{+\cdot\cdot} - u_{\cdots}]\delta^{+\cdot\cdot} + \\
&\quad 4d^{11}_{\cdots}[u_{-\cdot\cdot} - u_{\cdots}]\delta^{-\cdot\cdot} + \\
&\quad 4d^{22}_{\cdots}[u_{\cdot+\cdot} - u_{\cdots}]\delta^{\cdot+\cdot} + \\
&\quad 4d^{22}_{\cdots}[u_{\cdot-\cdot} - u_{\cdots}]\delta^{\cdot-\cdot} + \\
&\quad 4d^{33}_{\cdots}[u_{\cdot\cdot+} - u_{\cdots}]\delta^{\cdot\cdot+} + \\
&\quad 4d^{33}_{\cdots}[u_{\cdot\cdot-} - u_{\cdots}]\delta^{\cdot\cdot-} + \\
&\quad [(d^{11}_{+\cdot\cdot} - d^{11}_{-\cdot\cdot})\delta^{\pm\cdot\cdot} + (d^{21}_{\cdot+\cdot} - d^{21}_{\cdot-\cdot})\delta^{\cdot\pm\cdot} + (d^{31}_{\cdot\cdot+} - d^{31}_{\cdot\cdot-})\delta^{\cdot\cdot\pm}](u_{+\cdot\cdot} - u_{-\cdot\cdot})\delta^{\pm\cdot\cdot} + \\
&\quad [(d^{12}_{+\cdot\cdot} - d^{12}_{-\cdot\cdot})\delta^{\pm\cdot\cdot} + (d^{22}_{\cdot+\cdot} - d^{22}_{\cdot-\cdot})\delta^{\cdot\pm\cdot} + (d^{32}_{\cdot\cdot+} - d^{32}_{\cdot\cdot-})\delta^{\cdot\cdot\pm}](u_{\cdot+\cdot} - u_{\cdot-\cdot})\delta^{\cdot\pm\cdot} + \\
&\quad [(d^{13}_{+\cdot\cdot} - d^{13}_{-\cdot\cdot})\delta^{\pm\cdot\cdot} + (d^{23}_{\cdot+\cdot} - d^{23}_{\cdot-\cdot})\delta^{\cdot\pm\cdot} + (d^{33}_{\cdot\cdot+} - d^{33}_{\cdot\cdot-})\delta^{\cdot\cdot\pm}](u_{\cdot\cdot+} - u_{\cdot\cdot-})\delta^{\cdot\cdot\pm} + \\
&\quad 2d^{12}_{\cdots}(u_{++\cdot} - u_{+-\cdot})\delta^{+\pm\cdot} + \\
&\quad 2d^{12}_{\cdots}(u_{--\cdot} - u_{-+\cdot})\delta^{-\pm\cdot} + \\
&\quad 2d^{13}_{\cdots}(u_{+\cdot+} - u_{+\cdot-})\delta^{+\cdot\pm} + \\
&\quad 2d^{13}_{\cdots}(u_{-\cdot-} - u_{-\cdot+})\delta^{-\cdot\pm} + \\
&\quad 2d^{23}_{\cdots}(u_{\cdot++} - u_{\cdot+-})\delta^{\cdot+\pm} + \\
&\quad 2d^{23}_{\cdots}(u_{\cdot--} - u_{\cdot-+})\delta^{\cdot-\pm} \\
&\Big\}
\end{aligned}
\tag{4.35}
$$

where, e.g., $d^{11}_{\cdot+\cdot}$ is element $\{1,1\}$ of the diffusion tensor at point $\{i, j+i, k\}$ and δ is defined as 1 for tissue points and 0 for void points. For example,

$$
\delta^{\cdot+\cdot} = \begin{cases} 1 & \text{if } \{i, j+1, k\} \text{ is tissue} \\ 0 & \text{if } \{i, j+1, k\} \text{ is void.} \end{cases} \tag{4.36}
$$

Delta functions with indices including $\pm$ test both neighbours on the axis. For example, $\delta^{\cdot\pm\cdot} \equiv \delta^{\cdot+\cdot}\delta^{\cdot-\cdot}$.

4.11.1 Anisotropy in Two Dimensions

One fortunate corollary of using delta functions to exclude void neighbours is that two-dimensional meshes can be accommodated with no modification. Absence of neighbours in the z direction will simply eliminate diffusion along the z axis.

When running a two-dimensional simulation, Beatbox preserves all three components of the fibre vector, without 'flattening' the vector to two dimensions. This prevents problems with automatic normalisation where fibres lie parallel to the z axis. This will also allow users to run simulations on two-dimensional slices of a mesh using three-dimensional fibre data.

4.11.2 Implementation

The implementation of the above equations in Beatbox's `diff` device is broken into two parts — precomputation of $\mathbf{D}$ and other constants for each tissue point, and regular computation of the Laplacian.

Precomputation of per-point constants takes place in the `diff` device's create function. As a collection of values must be stored for each tissue point, `diff` first counts the number of tissue points and allocates memory for an array of `AnisoPoint` structures, a reference to which is held by the device's `STR` structure. The structures, defined in Table 4.2, hold the coordinates of the point in question, along with its local diffusion tensor and selected differences of the diffusion tensor.

Type	Name	Description
`int`	`x`	x coordinate of the point.
`int`	`y`	y coordinate of the point.
`int`	`z`	z coordinate of the point.
`real`	`D[][]`	Diffusion tensor.
`real`	`c[]`	Differences of diffusion coefficients.

Table 4.2: Fields of the `Anisopoint` structure.

The diffusion tensor, $\mathbf{D}$, is precomputed according to (4.11). Code for the operation is shown in Listing E.4.

To improve the performance of computations during the simulation run, the differences of $\mathbf{D}$ used in (4.35) are also precomputed, as shown in (4.37–4.39). The c values are stored in the `c` array of each `Anisopoint` structure. Code for the operation in shown in Listing E.5.

$$c_1 = [(d^{11}_{+\cdot\cdot} - d^{11}_{-\cdot\cdot})\delta^{\pm\cdot\cdot} + (d^{21}_{\cdot+\cdot} - d^{21}_{\cdot-\cdot})\delta^{\cdot\pm\cdot} + (d^{31}_{\cdot\cdot+} - d^{31}_{\cdot\cdot-})\delta^{\cdot\cdot\pm}] \tag{4.37}$$

$$c_2 = [(d^{12}_{+\cdot\cdot} - d^{12}_{-\cdot\cdot})\delta^{\pm\cdot\cdot} + (d^{22}_{\cdot+\cdot} - d^{22}_{\cdot-\cdot})\delta^{\cdot\pm\cdot} + (d^{32}_{\cdot\cdot+} - d^{32}_{\cdot\cdot-})\delta^{\cdot\cdot\pm}] \tag{4.38}$$

$$c_3 = [(d^{13}_{+\cdot\cdot} - d^{13}_{-\cdot\cdot})\delta^{\pm\cdot\cdot} + (d^{23}_{\cdot+\cdot} - d^{23}_{\cdot-\cdot})\delta^{\cdot\pm\cdot} + (d^{33}_{\cdot\cdot+} - d^{33}_{\cdot\cdot-})\delta^{\cdot\cdot\pm}] \tag{4.39}$$

By storing `Anisopoint` structures for tissue points only, `diff` can more efficiently traverse the mesh at runtime. Rather than iterating over the mesh, testing point status in `Geom` at each location, `diff` can iterate linearly over the `tissuePoints` array. Not only will this prevent the repeated testing of points' geometry status, but accessing a linear data structure will most likely improve processor cache performance.

In its run function, `diff` iterates over the list of tissue points, and computes the Laplacian at each point using fields from the `Anisopoint` structure. The anisotropic branch of the `diff` device's run function is shown in Listing E.6.

4.11.3 Script Interface

The user can turn anisotropy on in the script by setting the `state` module's `anisotropy` parameter to `1`. As third party device developers may wish to alter the behaviour of their devices on an anisotropic domain, the status of anisotropy is made global, via the `ANISOTROPY_ON` variable in *state.h*.

When anisotropy is off, the `diff` device requires a scalar diffusion coefficient, D, supplied as the `D` parameter. With anisotropy on, `diff` requires two diffusion coefficients, $D_\parallel$ and $D_\perp$, supplied as the `Dpar` and `Dtrans` parameters respectively. With anisotropy on, the isotropic diffusion coefficient will be ignored, and vice-versa.

4.12 Verification

The implementation of anatomically realistic geometry was tested with a series of simulations using either a simplified model of kinetics, `fhncub` [FitzHugh, 1961], and a biophysically realistic model, `lrd` [Luo and Rudy, 1994b].

Simulation Parameters

For the FitzHugh-Nagumo simulations, $\Delta t = 0.005$, $\Delta x = 0.3\dot{3}$; the isotropic diffusion coefficient, $D = 1.0$; anisotropic coefficients $D_{\parallel} = 1$ and $D_{\perp} = D_{\parallel}/4$[as in Aslanidi et al., 2006]. For the Luo-Rudy simulations, $\Delta t = 0.005$, $\Delta x = 0.1$; the isotropic diffusion coefficient, $D = 0.3125$[as in Biktasheva et al., 2005]; anisotropic coefficients $D_{\parallel} = 0.3125$ and $D_{\perp} = D_{\parallel}/4$.

Four geometry files are used throughout the verification simulations:

square.bbg Describes a 50×50 mesh. All points have fibre vectors of $\{1, 0, 0\}$.

cube.bbg Describes a $50 \times 50 \times 50$ mesh. All points have fibre vectors of $\{1, 0, 0\}$.

ffr.bbg three-dimensional mesh describing rabbit ventricles, measured by Vetter and McCulloch [1998] and discretised for finite differences by Fenton et al. [2005]. Containing box is $117 \times 107 \times 129$ (1,614,951 points), with 470,324 tissue points. Tissue shape and fibre directions obtained by NMR.

ffr_yslice.bbg two-dimensional slice of *ffr.bbg* at $y = 50$.

Reentrant waves are initiated using two methods. In FitzHugh-Nagumo kinetics, a cross-field stimulation protocol is used, where the mesh is divided into quarters, with differing initial states of u and v in each quadrant:

Top Left $u = 1.7, v = 0.7$

Top Right $u = -1.7, v = 0.7$

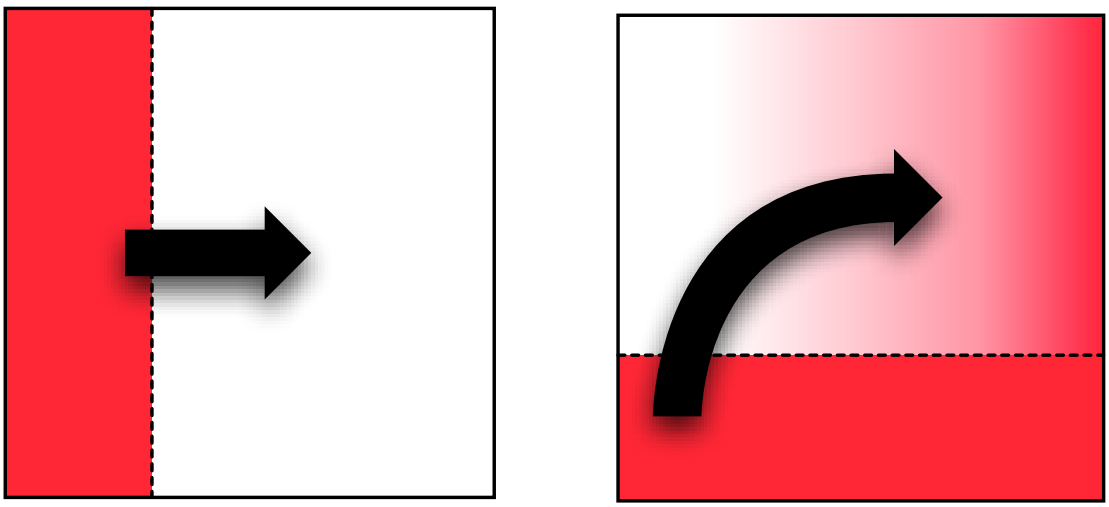
Bottom Left $u = 1.7, v = -0.7$

Bottom Right $u = -1.7, v = -0.7$

For Luo-Rudy, an S1–S2 protocol is used, as shown in Figure 4.9 in which two waves are initiated, perpendicular to one another, and with a time delay. As the wave initiated by the first stimulus moves from left to right, the second stimulus is applied to the lower half of the medium. The resulting second wave refracts around the tail of the first, forming a spiral.

To test isotropic diffusion in the abstract, simulations were run using simple cuboid meshes and compared to those using an identical meshes described in Beatbox geometry files (*square.bbg* and *cube.bbg*). Automated comparisons of output files showed that both approaches produced identical results. Excitation was seen to propagate uniformly, which is as expected for isotropic diffusion. Simulations using two- and three-dimensional geometry files, with and without anisotropy, were run sequentially and using MPI. Automatic comparison of output files showed that simulations run in parallel produced identical results to those run sequentially.

Anisotropic simulations show wavefronts to be elongated along the fibre direction, which is as expected given the higher longitudinal diffusion coefficient. The conduction velocity is measured using two instances of the `poincare` device on both the x and y axes. Wavefronts are detected as positive crossings of the transmembrane voltage variable (`u` for FitzHugh-Nagumo and `V` for Luo-Rudy) relative to their respective 'mid' values (`0.0` for FHN and `-40.0` for LRd). Conduction



(a) S1 (b) S2

Figure 4.9: S1–S2 stimulation protocol.

velocity in the fibre direction was measured to be approximately equal to that achieved in an isotropic mesh, where $D = D_{\parallel}$. Orthogonal to the fibres, conduction velocity is approximately equal to an isotropic mesh with $D = D_{\perp}$.

Realistic 2D Mesh

Interaction with the irregular boundary can be examined on a two-dimensional slice of an anatomically realistic mesh (*ffr_yslice.bbg*). Figure 4.10 shows the fibre directions in the geometry file. Figures 4.11 and 4.12 show the initiation of a spiral wave in the slice. Wavefronts are absorbed by the boundaries as expected for no-flux boundary conditions. Furthermore, waves reaching the boundary show no signs of reflection, and tissue at rest is not excited by the presence of the boundary.

Realistic 3D Mesh

As Figures 4.14 and 4.15 show, in three dimensions, the boundary conditions continue to absorb waves as expected, and do not cause excitation in resting tissue. In addition, wavefronts are distorted by variances in conduction velocity, as expected. This can be seen by comparing, e.g. item (i) of each figure where the isotropic figure shows thick, solid waves throughout the tissue, while the wavefronts in anisotropic tissue vary in thickness through the tissue wall.

All verification simulations show expected behaviour throughout. When run in parallel, produce identical results to those run sequentially. It can be concluded that the methods described above are successful in modelling anisotropic diffusion and no-flux boundary conditions in regular and with anatomically realistic tissue geometry.

4.13 Geometry Performance with MPI

The performance of Beatbox's geometry features was tested using a set of three simulations, each of which were run with FitzHugh-Nagumo and Luo-Rudy kinetics on 1–512 processes, using the

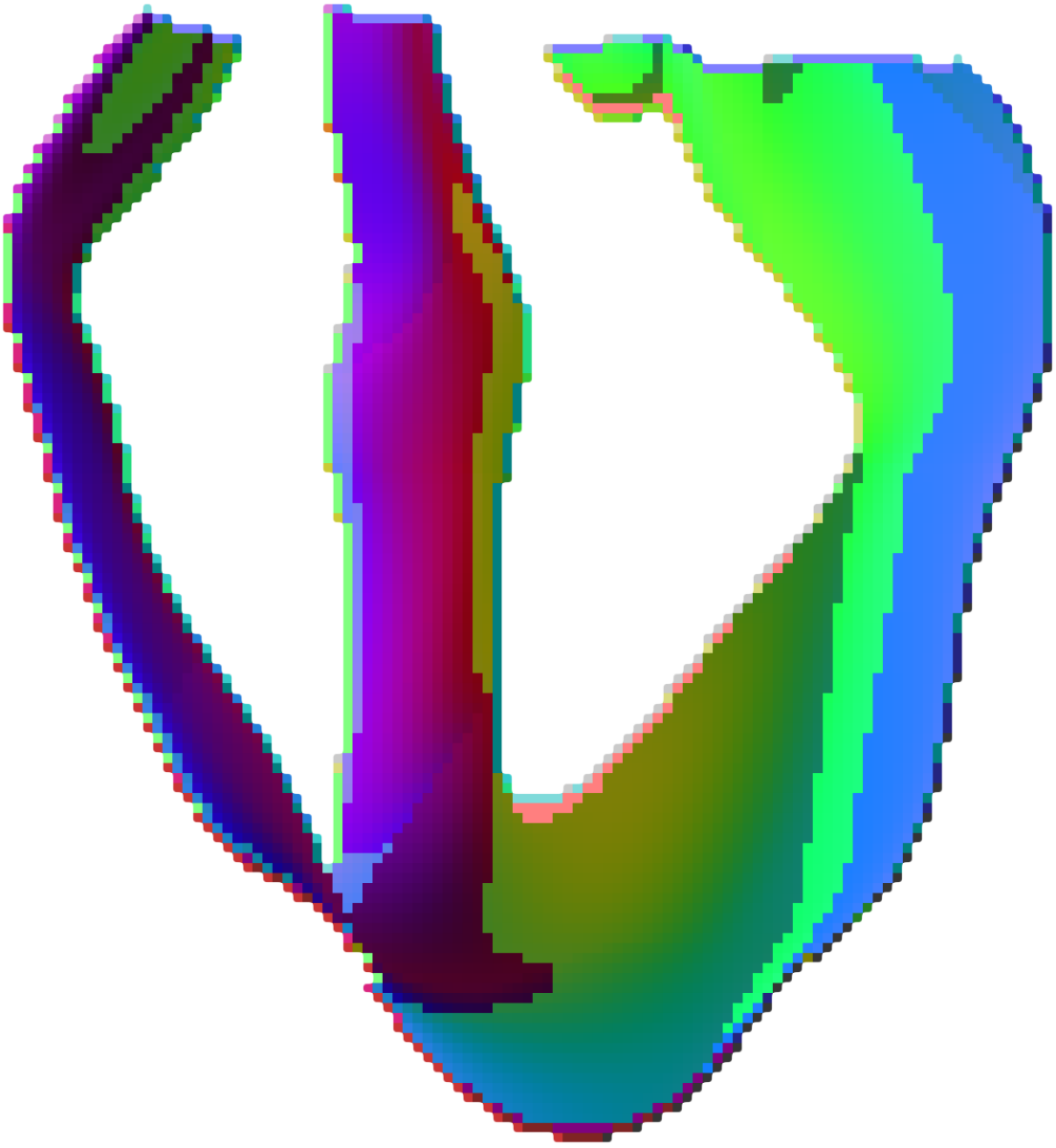
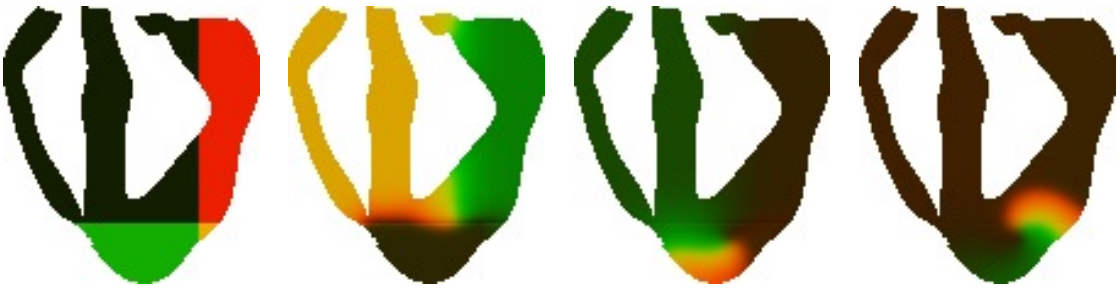


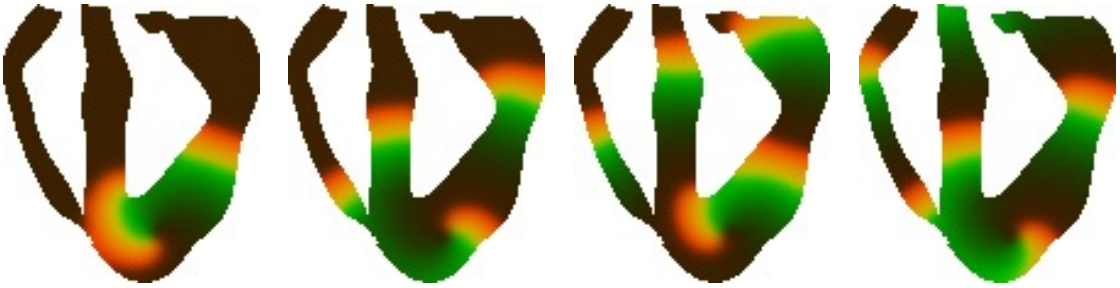
Figure 4.10: Colour mapping of the fibre directions in the `ffr_yslice.bbg` geometry file. x, y and z fibre components are mapped to red, green, and blue respectively.

HECToR Phase 2a machine described in Section 3.15. The simulations centred around a geometry file describing rabbit ventricles. The mesh contains 1,699,201, of which 470,324 are tissue. All three scripts contained `diff`, `euler`, `ppmout`, `record` and `dump` devices, all of which ran on every timestep. Code for all three scripts is given in Appendix F.

Script 1 Cuboid mesh containing approximately the same number of tissue points as the rabbit ventricle mesh ($69 \times 80 \times 92$). Isotropic diffusion.



(a) $t = 0$ (b) $t = 1000$ (c) $t = 2000$ (d) $t = 3000$



(e) $t = 4000$ (f) $t = 5000$ (g) $t = 6000$ (h) $t = 7000$

Figure 4.11: `geometry=ffr_yslice.bbg`, `anisotropy=0`, `rhs=fhncub`, cross-field stimulus. The colour scale used is shown in Figure 4.13. The Beatbox script for this simulation is shown in Listing F.1.

Script 2 Rabbit ventricle mesh ($119 \times 109 \times 131$). Isotropic diffusion.

Script 3 Rabbit ventricle mesh ($119 \times 109 \times 131$). Anistropic diffusion.

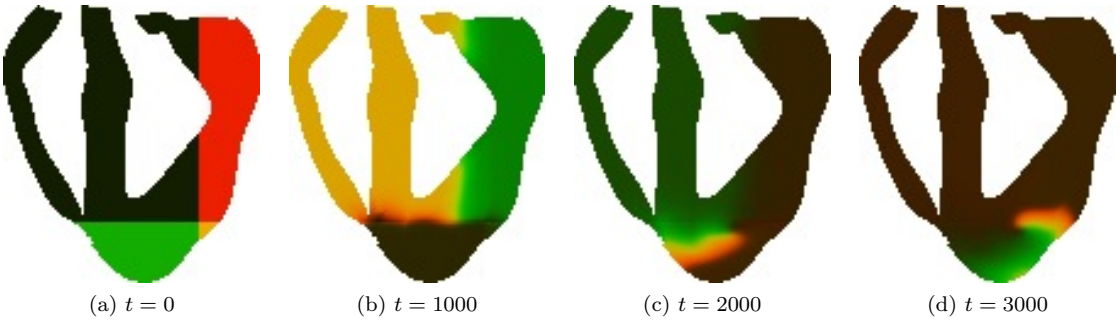
Due to the differing sizes of the containing meshes, the domain decompositions differ above 64 processes. Table 4.3 details the decompositions of the two meshes used.

4.13.1 `euler` Scaling Performance

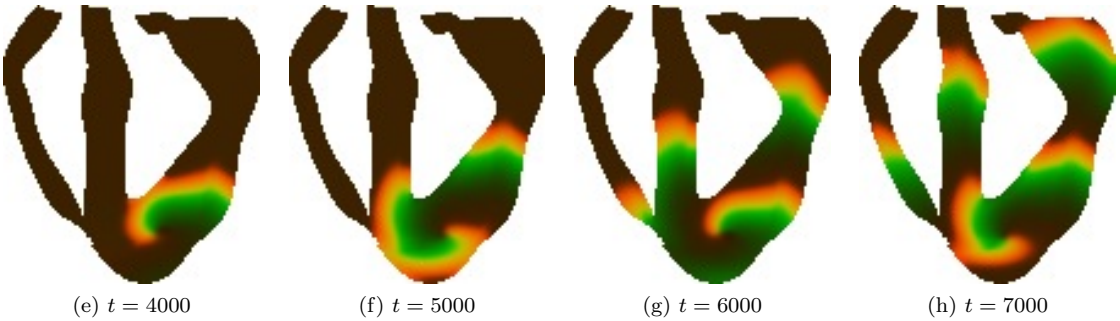
Figure 4.16 compares the scaling performance of the `euler` device for each of the scripts with FitzHugh-Nagumo and Luo-Rudy kinetics. In both figures, we see very similar performance of `euler` with Scripts 2 and 3. This shows that anisotropy has no effect on this device, which is what should be expected, as `euler` does not run at all differently when the anisotropy feature is active.

As can be seen from both figures, Script 1 significantly outperforms Scripts 2 and 3. While some reduction in performance can be attributed to time spent iterating over the sparse mesh, querying `Geom` at each point, the primary cause is one of load balance. Figure 4.17 shows the load balance of the rabbit ventricle mesh when decomposed by Beatbox. As the processor count increases, the number of processors doing comparatively little work increases, thus degrading the scaling performance. This strongly suggests that the domain decomposition for cuboid meshes does not extend well to anatomically realistic geometry.

Comparing Figure 4.16a with 4.16b, it can be seen that Script 1 performs significantly better with Luo-Rudy kinetics. As discussed in Section 3.15.1, `euler` performs better when using a more processor-intensive RHS module, such as Luo-Rudy. Contrary to this, Scripts 2 and 3 see a



(a) $t = 0$ (b) $t = 1000$ (c) $t = 2000$ (d) $t = 3000$



(e) $t = 4000$ (f) $t = 5000$ (g) $t = 6000$ (h) $t = 7000$

Figure 4.12: `geometry=ffr_yslice.bbg`, `anisotropy=1`, `rhs=fhncub`, cross-field stimulus. The colour scale used is shown in Figure 4.13. The Beatbox script for this simulation is shown in Listing F.2.

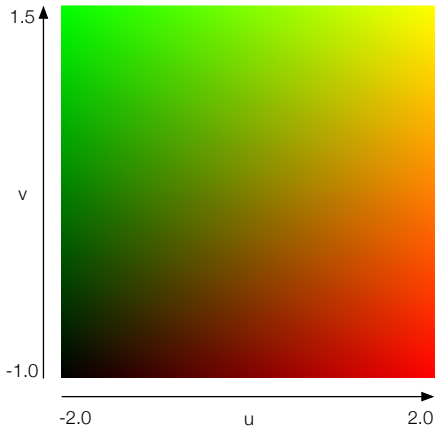


Figure 4.13: Colour scale for Figures 4.11 and 4.12.

	Decomposition			
	Cuboid Mesh		Rabbit Mesh	
Total Processes	Active Processes	Decomposition	Active Processes	Decomposition
1	1	$(1 \times 1 \times 1)$	1	$(1 \times 1 \times 1)$
4	4	$(1 \times 2 \times 2)$	4	$(1 \times 2 \times 2)$
8	8	$(2 \times 2 \times 2)$	8	$(2 \times 2 \times 2)$
16	16	$(2 \times 2 \times 4)$	16	$(2 \times 2 \times 4)$
32	32	$(2 \times 4 \times 4)$	32	$(2 \times 2 \times 4)$
64	64	$(4 \times 4 \times 4)$	64	$(4 \times 4 \times 4)$
128	128	$(4 \times 8 \times 4)$	126	$(7 \times 6 \times 3)$
256	224	$(4 \times 8 \times 7)$	252	$(7 \times 6 \times 6)$
512	416	$(4 \times 8 \times 13)$	504	$(7 \times 9 \times 8)$

Table 4.3: Decompositions of scripts 1, 2 and 3.

slight reduction in performance with Luo-Rudy. Again, load imbalance is the likely cause, with the increased work per point exacerbating the disparity between processes.

4.13.2 `diff` Scaling Performance

Figure 4.18 compares the scaling performance of the `diff` device for Scripts 1, 2 and 3 using FitzHugh-Nagumo and Luo-Rudy kinetics.

As discussed in Section 3.15.2, the performance of `diff` is largely characterised by communication via `haloSwap()`, so the effect of geometry can not be seen as clearly. As `haloSwap()` exchanges all points on the subdomain's surfaces, not just tissue points, the differences between Scripts 1 and 2 are likely to be related to the differing mesh sizes, and thus less data to be sent and received per process. At larger process counts, the advantage of the smaller mesh appears to subside somewhat, either due to the diminishing surface area of processes, or because the time taken to exchange data is small in relation to the time required to establish connections with neighbouring processes.

Using the anisotropy feature approximately doubles the average runtime of `diff`. Since no additional information is being exchanged when using anisotropy, this difference can be attributed to the extra computational load of anisotropy, and any associated increase in synchronisation time caused by the load imbalance discussed above. With both FitzHugh-Nagumo and Luo-Rudy kinetics, Script 3 performs best at larger processor counts. While the cause of this is not clear from the data given, it could be conjectured that the additional work required by `diff` for anisotropic diffusion shifts the balance of computation and communication, allowing the device to benefit from additional processors.

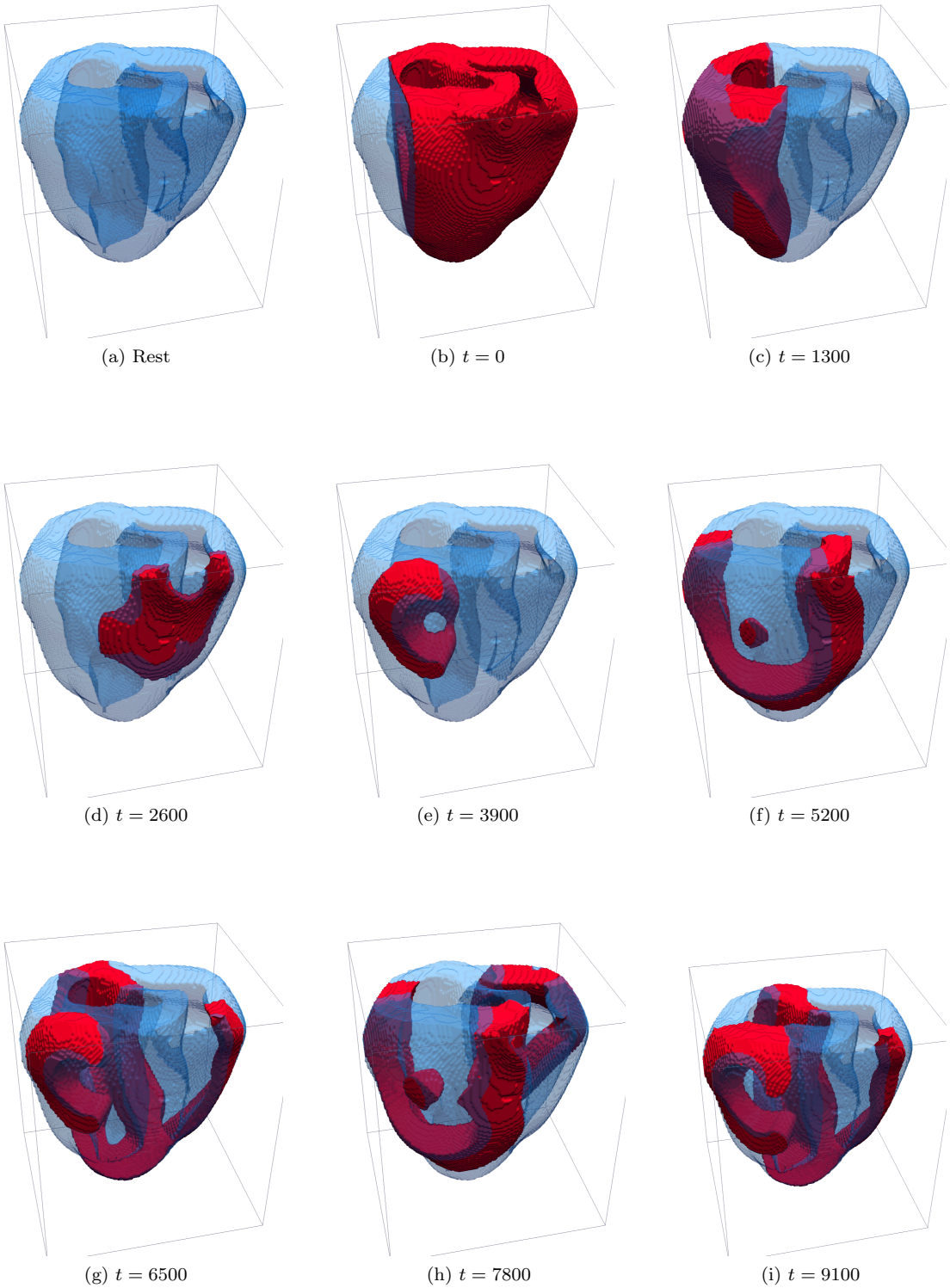
4.14 Extending Beatbox with Geometry

Third-party devices that do not take account of anatomical features of the mesh will be unaffected by the addition of the features described in this chapter. Devices that perform computation at each point in the mesh can improve their performance by testing the status of each point under consideration using `isTissue()`, but this is not required for the device to operate, and does not affect the parallel safety of the device. Devices that do make use of `Geom` will have to ensure that they access only locally available coordinates, as with `New`, by limiting accesses to their

local space $(x \pm 1, y \pm 1, z \pm 1)$. In addition, as the description of tissue geometry is assumed to be static and thus is not synchronised between processes, `Geom` must be treated as read-only to ensure parallel safety.

Consequently, the rules for device development are modified as follows, with additions shown in bold type:

1. A device must only alter layers of `New` in its space, at points in its space.
2. All operations on `New` must be commutative — the order of operation cannot be assumed.
3. Aside from the permitted regions of `New`, a device may only alter the values of local variables or those in its parameter structure.
4. Assignments made to a device's parameter structure must not be derived from data in `New`.
5. **Assignments made to a device's parameter structure must not be derived from data in `New`.**
6. Assignments made to a device's parameter structure must not be derived from data in its `Space` structure.
7. Assignments made to a device's parameter structure must not be derived from local minima or maxima (e.g. `local_xmax`).
8. Assignments made to a device's parameter structure must not be derived from a random number generator.
9. From any point in its space, a device may not reference points in `New` beyond $\{x \pm 1, y \pm 1, z \pm 1\}$.
10. A device that references neighbouring points in `New` must put the `DEVICE_REQUIRES_SYNC` macro in its Create function.
11. **From any point in its space, a device may not reference points in `Geom` beyond $\{x \pm 1, y \pm 1, z \pm 1\}$.**
12. Files accessed from a device must only be read.

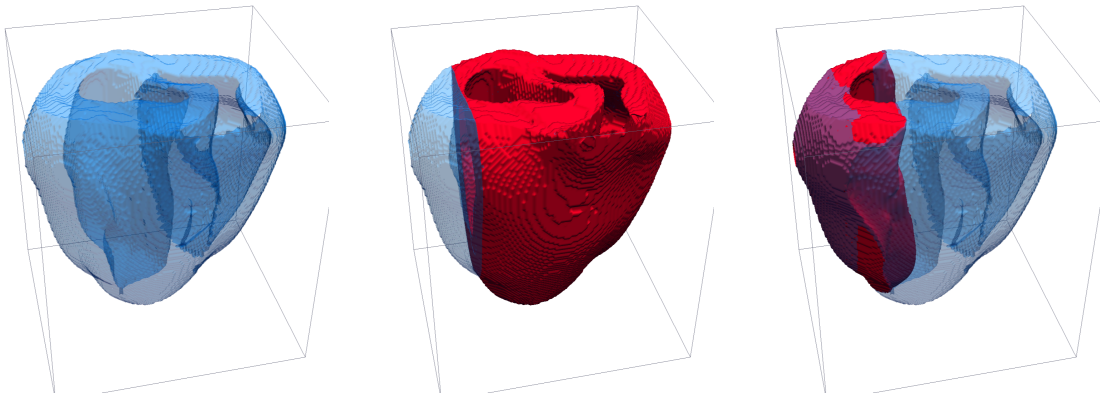


(a) Rest (b) $t = 0$ (c) $t = 1300$

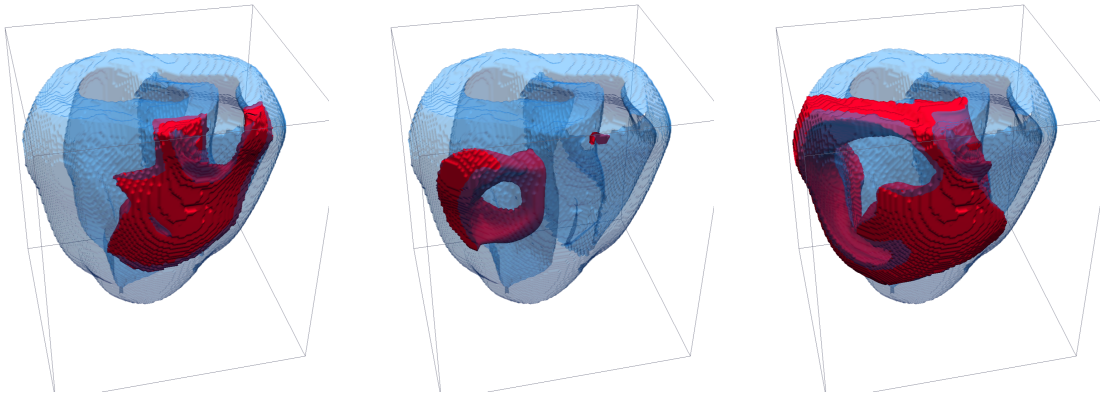
(d) $t = 2600$ (e) $t = 3900$ (f) $t = 5200$

(g) $t = 6500$ (h) $t = 7800$ (i) $t = 9100$

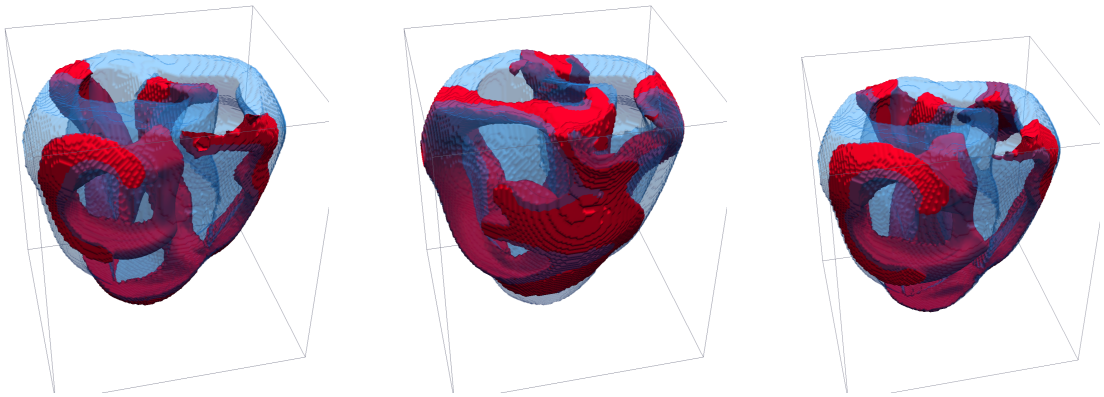
Figure 4.14: `geometry=ffr.bbg`, `anisotropy=0`, `rhs=fhncub`, cross-field stimulus. $\Delta x = 0.333$, $\Delta t = 0.005$, D = 1. The visualisation, produced using Paraview, shows the tissue surface (translucent blue) as an isosurface at 0.5 in the tissue status field. Wavefronts (solid red) are visualised as an isosurface in `u`. The Beatbox script for this simulation is shown in Listing F.3.



(a) Rest (b) $t = 0$ (c) $t = 1300$

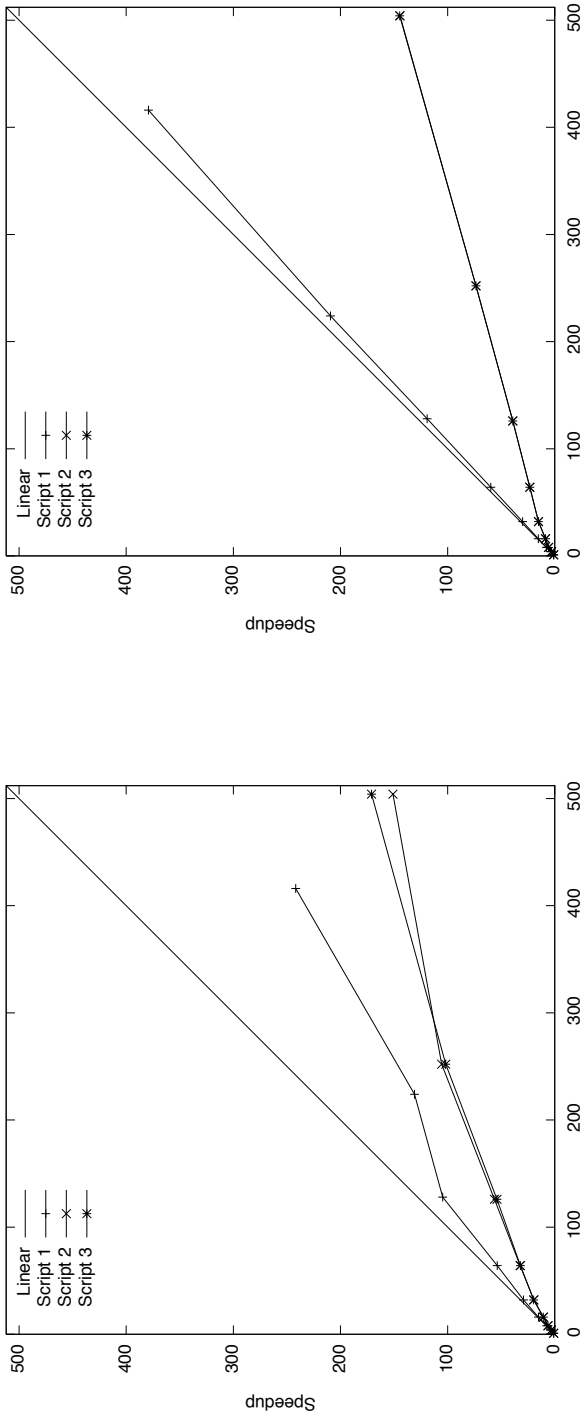


(d) $t = 2600$ (e) $t = 3900$ (f) $t = 5200$



(g) $t = 6500$ (h) $t = 7800$ (i) $t = 9100$

Figure 4.15: `geometry=ffr.bbg`, `anisotropy=1`, `rhs=fhncub`, cross-field stimulus. $\Delta x = 0.333$, $\Delta t = 0.005$, $D_{\parallel} = 1$, $D_{\perp} = D_{\parallel}/4$. The visualisation, produced using Paraview, shows the tissue surface (translucent blue) as an isosurface at 0.5 in the tissue status field. Wavefronts (solid red) are visualised as an isosurface in `u`. The Beatbox script for this simulation is shown in Listing F.4.

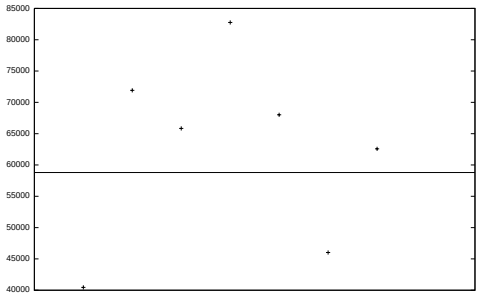


(a) FitzHugh-Nagumo kinetics

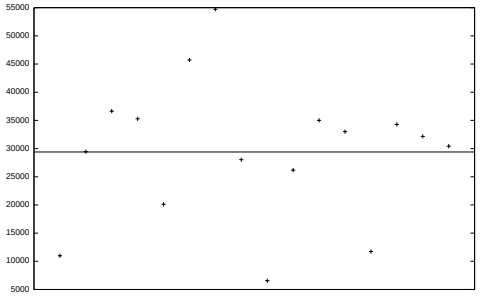
(b) Luo-Rudy kinetics

Figure 4.16: Scaling performance of the `euler` device running scripts 1,2 and 3.

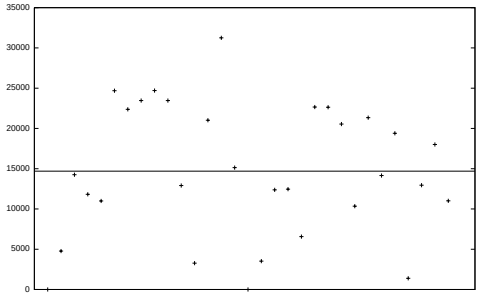
(a) 4 processes



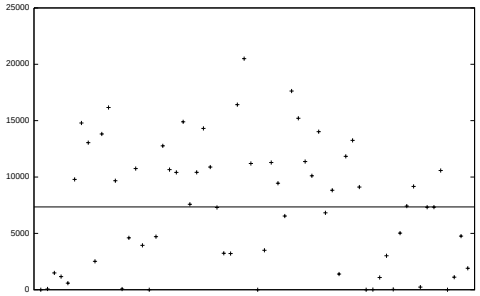
(b) 8 processes



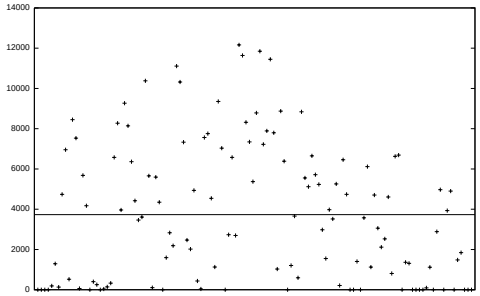
(c) 16 processes



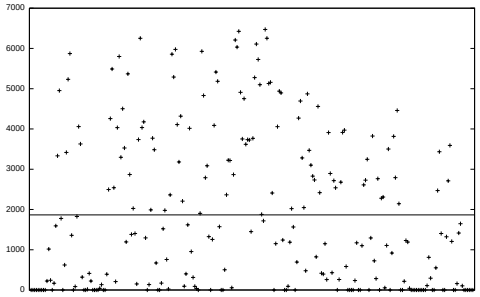
(d) 32 processes



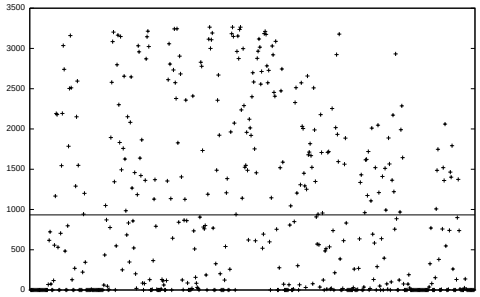
(e) 64 processes



(f) 128 processes

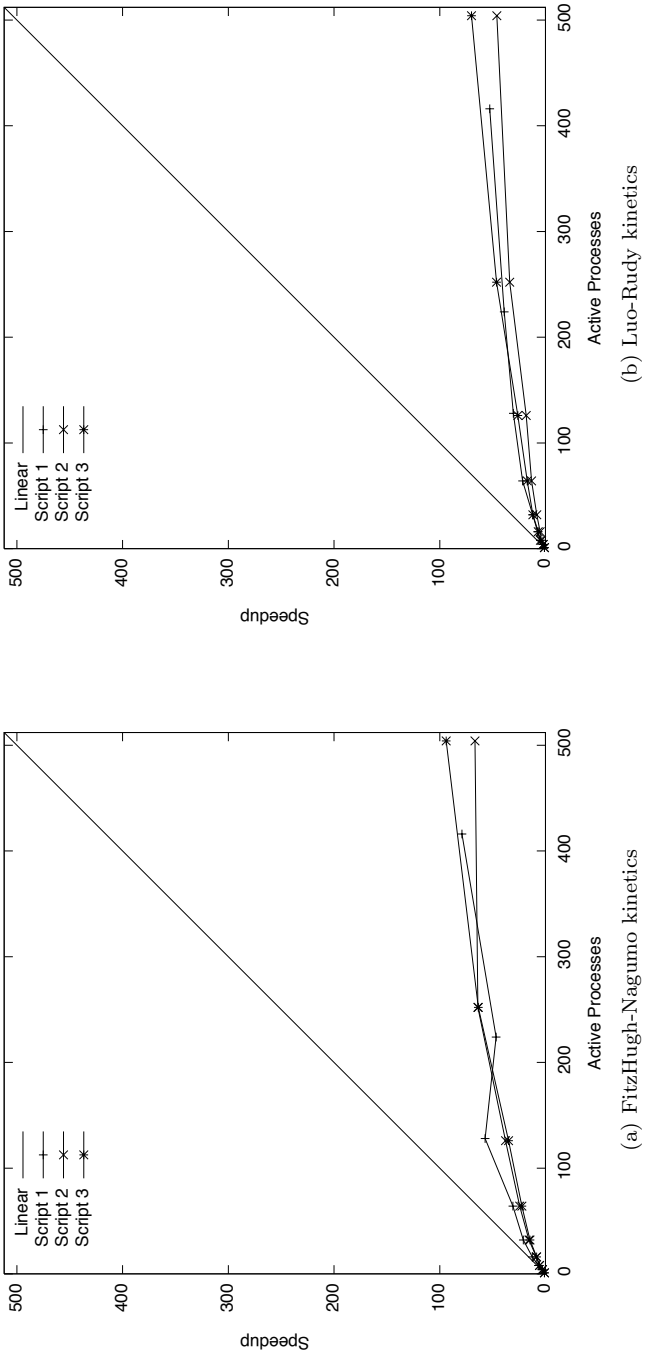


(g) 256 processes



(h) 512 processes

Figure 4.17: Load balance of the rabbit ventricle mesh used in Scripts 2 and 3, when decomposed by Beatbox. Charts show the number of tissue points per process. Horizontal lines show the mean of tissue points in the mesh per process. Ideal load balancing would show all points on the mean. The load balancing shown in these tests is considered poor.



(a) FitzHugh-Nagumo kinetics

(b) Luo-Rudy kinetics

Figure 4.18: Scaling performance of the `diff` device running scripts 1,2 and 3.

4.15 Summary

In this chapter, we have seen Beatbox's meshes extended to include descriptions of anatomically realistic geometry. We have discussed the user interface to the geometry feature, and proposed a simple file structure describing anatomically realistic meshes.

Two methods for computing Neumann conditions on the irregular boundary were considered. The chosen scheme integrates boundary conditions with the computation of the Laplacian using delta functions.

We saw the verification of Beatbox's realistic tissue geometry from simulations in two- and three-dimensional isotropic and anisotropic meshes. Boundary conditions were seen to be effective. We identified poor load balancing when anatomically realistic meshes are parallelised, due to the varying number of tissue points in each process subdomain. Further development of Beatbox should investigate, as a matter of urgency, adaptive domain decomposition based on the number of tissue points in each subdomain.

The software implementation of the anatomically realistic meshes was discussed in detail, and modifications to affected devices were described. Finally, we revised the 'laws of devices' in light of these new features.

CHAPTER

FIVE

CONCLUSIONS

5.1 Parallelism

The QUI scripting language is a domain-specific language for cardiac modelling. The device abstraction allows users to structure simulations from familiar units of functionality, which can be parameterised as desired. The range of available devices provides users with the ability to stimulate, measure and record activity in the simulation medium. In addition, a broad selection of RHS modules offers users choice in the level of physiological realism and computation efficiency of their simulations. The `k_func` device allows users to customise simulations further from within the script, by defining functions to be executed at runtime. User-defined functions may operate on the simulation medium itself, or on global k_variables defined in the script.

User-defined functions interpreted by the `k_func` device have been constrained to ensure parallel safety. A function will reliably produce the same results sequentially and in parallel if no local value is assigned to a global variable; no references are made to points beyond $\{x \pm 1, y \pm 1, z \pm 1\}$; and operations performed on local data are commutative. The `k_func` device has been adapted to ensure that all user-defined functions meet these criteria. Where necessary, new parallelised devices have been added to replace some of the operations no longer allowed by `k_func`.

The parallelised QUI package, renamed Beatbox, provides a similar set of functionality, capable of being run on distributed memory machines using the MPI standard. Beatbox's parallel implementation gives its users scope to design and run far larger simulations than were possible when bound to a single machine using QUI. Beatbox's adaptation of the QUI scripting language abstracts its parallel implementation from users, maintaining the illusion of running sequentially. Any valid Beatbox script is parallel-safe and capable of producing identical results sequentially or in parallel. Users play no part in, and require no knowledge of domain decomposition; the domain is partitioned and shared amongst the available processes automatically. Where necessary, the domain decomposition process will sacrifice processes in order to reduce communication. Sequential and parallel versions alike take the same input parameters and files, and produce identical outputs in both content and format. The only distinguishing factor between simulations run sequentially and in parallel is the time taken to complete them.

5.2 Parallel Performance

Beatbox's core computational devices, `euler` and `diff`, have been shown to scale very efficiently on up to 256 processes, with scope for scaling to many more. Useful scaling of Input/Output devices is limited to between 64 and 128 processes, beyond which scaling regresses due to contention for access to the file server. Of the available output devices, binary files produced by `dump` offer the most efficient means of recording the results of a simulation.

Overall performance is best when using computationally complex RHS modules, as doing so reduces shifts the balance of scalable computation to poorly scaling I/O. This bodes well for large-scale, biophysically realistic simulations. Any future extensions of Beatbox, such as the introduction of bidomain tissue models, would likely improve this balance further.

Scaling performance of the `diff` device does not appear to be affected by the total volume of data to be written. From this, it is reasonable to suggest that increased mesh sizes should also produce similar I/O scaling performance and benefit from further improvements in the computation-to-I/O ratio.

At present, Beatbox is best suited to simulations involving computationally complex RHS models on large meshes. Output should be as infrequent as possible, preferably using `dump`.

Of the current state of the art software packages, both Carp and Chaste have undergone significant improvements in parallel performance in the last year or so. The authors of Chaste claim effective scaling to 2048 processes [University of Oxford Computing Laboratory, 2010], a number that is subject to ongoing improvement. A recent Cray Centre of Excellence programme has extended the scalability of Carp from around 512 cores to 16,000 [Plank et al., 2010]. In both cases, significant efforts were made to improve load balancing in the domain decomposition, and the efficiency of I/O. Both of these areas would benefit from further development in Beatbox.

In some respects, however, it is unfair to compare scaling in these terms. The scalability of a problem can often be an indication of its initial difficulty. The efficiency of Beatbox's computational methods may limit scalability for relatively easy problems. A true comparison of performance can only be formed with side-by-side execution of the same simulation.

5.3 Anatomically Realistic Tissue Geometry

Beatbox has been extended to allow descriptions of anatomically realistic tissue geometry. Geometries are described in separate files, which are simple to create and forgivingly interpreted by Beatbox. The addition of the geometry feature does not affect existing scripts and allows geometries to be interchanged with relative ease. Boundary conditions are computed in conjunction with the Laplacian using delta functions, and have been shown to be stable and absorb oncoming waves as expected.

Load imbalance caused by anatomically realistic meshes adversely affects parallel performance, due to the varying number of tissue points in each process subdomain. We propose further investigation into adaptive domain decomposition schemes as a matter of urgency.

5.4 Usability

Users familiar with QUI's scripting interface will find relatively little changed, although the majority of QUI scripts will require some modification in order to be used with Beatbox. For new users, Beatbox's scripting language should be relatively easy to pick up; its similarity to other programming languages will assist those with a computing background, while its minimal, domain-specific scope should mean that programming novices won't struggle unduly.

Some of the limitations imposed to maintain parallel safety may raise surprising errors for new users; in particular, problems concerning the `nowhere` parameter have the potential to confuse. By preventing users from running scripts that will not be parallel-safe, Beatbox obviates the need for painstaking debugging that would intimidate most users. Instead, clear error messages explain the problem to the user and, if necessary, ask them to try again. Supported by approachable documentation, users should be able to quickly overcome these hurdles and use the software confidently.

Wherever possible users can write scripts freely without any consideration for whether the simulation will be run sequentially or in parallel. To satisfy the needs of a small number of existing QUI users, some compromises were made, which give rise to a small inconsistency. Some real-time visualisation devices, which cannot be usefully parallelised, are conditionally enabled in the sequential version of the code. If a script using such a device is used in parallel, Beatbox will report that the device has been disabled, and that the simulation will continue without it. Since this only affects visualisation devices, the results of the simulation will be unaffected.

5.5 Software Construction

In comparison to the OpenMP parallelisation of QUI undertaken by Wolfe [Unpublished], Beatbox allows simulations to be scaled beyond the limitations of a single machine. The current memory usage of Beatbox justifies the stance taken to implement in a distributed memory environment only. Given current RAM capacities of nodes in state of the art machines such as HECToR and the rate of increase in cores per processor, it is likely to be some time before the reduction in RAM per node necessitates the use of mixed-mode programming.

QUI's modular structure forms a strong basis for parallelism in Beatbox. The commonality of devices allowed domain decomposition and space constraint to be handled centrally. Indeed, several device codes remain effectively 'unaware' of the parallel environment in which they are run.

5.5.1 Extensibility

Developers are able to extend Beatbox in a manner similar to QUI. The device and QPP APIs are well-defined and a suite of template and shortcut macros greatly simplify the development of new devices. As the majority of devices operate on the simulation medium by iterating over their space, many third-party devices will benefit from Beatbox's *implicit parallelism*. This should ensure that scaling performance is not unduly hampered by third-party code and the onset of Amdahl's Law.

Parallelism does introduce some complications for device developers. The 'laws of devices' have been developed throughout this thesis to aid developers in coding additional devices for Beatbox. The introduction of parallelism enforces the need for devices to adhere to the laws, and further laws are introduced to ensure that computations are valid. Third-party devices are largely unaffected by the addition of geometry. Devices wishing to make use of the feature, however, will find the API straightforward to use. Developing devices whose intended functionality falls outside the possibilities afforded by the Laws of Devices will still require specialist knowledge of parallelism and the MPI library, and is not recommended for inexperienced developers.

For all third-party devices, there is a question of trust. 'Hacky', inefficient or parallel-unsafe code has the ability to raise unnecessary errors or, worse, quietly produce incorrect results. In these circumstances, it is easy for users to blame the platform instead of the misbehaving third-party code. In the current architecture, there exists no ready means of establishing that

devices comply with the rules, or isolating the effect of devices that under-perform or behave unexpectedly. The object-oriented architecture and extensive body of test code used in the development of Chaste [Pitt-Francis et al., 2009] will likely help it to better withstand the trials of supporting third-party plugins, but this remains to be seen.

5.6 Further Work

5.6.1 Optimising Domain Decomposition

Central to Beatbox's domain decomposition process are the assumptions that each point in the mesh will require an equivalent amount of time for computation and, if exchanged between processes, an equivalent amount of time to communicate. These assumptions overlook the way in which users can specify the space of devices and, more importantly, the way in which anatomically realistic meshes will disable computation at many, if not most, points in the mesh. The consequent effect on load balance has been shown to significantly hamper performance. As a matter of priority, the means to more evenly partition anatomically realistic meshes should be developed. As discussed by Ridley [2010], tools for mesh partitioning are becoming available.

Less significant, but nonetheless important are the constants used to define the tradeoff between computation and communication. The current, heuristic, settings of these constants give intuitively acceptable decompositions, but are not based on the performance of the machine being used. The ability to test and adapt to the performance of the machine, for the current simulation, would allow Beatbox to make the most of the hardware.

5.6.2 Optimising Parallel I/O

One of the major limitation to Beatbox's scaling is that its runtime is dominated by I/O, for which scaling degrades past 64 or 128 processes. Parallel I/O places great demands on a machine's network, file servers and disks and is ultimately limited by them. To overcome the inherent limitations of parallel machines in this regard, it is necessary to isolate the scaling performance of I/O devices from those devoted to computation. This can be done by restricting I/O to a subset of processes, the optimum number of which will be machine-dependent.

5.6.3 Bidomain Modelling

Bidomain tissue models model the intra- and extra- cellular spaces as discrete continua. An increasing number of simulation studies are using bidomain tissue models, as they offer more accurate results for certain types of simulation study. While monodomain models characterise propagation in tissue accurately, bidomain models offer increased biophysical realism with regard to the effects of external stimuli. This is of particular interest to studies of defibrillation, for example. To satisfy the desire of the computational biology community to accurately model the anisotropy of the heart, bidomain functionality for Beatbox should be developed. The Finite Difference scheme employed by Beatbox should help to alleviate some of the implementation difficulties and inefficiencies experienced by others when solving elliptic problems on unstructured grids.

5.6.4 CellML Integration

The development of RHS modules from stand-alone codes is a well-documented and relatively straightforward process [M^{c}Farlane, 2007]. However, the process of producing code from pub-

lished models is laborious and often error-prone, as Hunter et al. [2006] describes:

> Typically, when a model is published in a scientific journal as a set of equations, any reader who wishes to use the information must write a simulation package and transcribe the equations from the printed text into computer code. This can be a difficult and time-consuming process because the published equations usually contain errors. Further, many biologists lack the ability or inclination to write the computer code needed to implement models, but they wish to use published models in the interpretation of their experimental data.

The CellML language has been developed to facilitate the structured and unambiguous description and exchange of cell models. The large body of models currently available in CellML is a compelling case for developing tools to automatically produce RHS modules from these XML-based descriptions.

5.6.5 Automatic Translation of MRI Meshes

The increasing availability of high-resolution MRI data describing tissue structure presents an exciting opportunity for Beatbox. As MRI datasets conform to regular meshes, tools to enable their translation into Beatbox geometry files could prove useful.

Segmented regions of tissue identified using MRI, such as areas containing different cell types, could be integrated into Beatbox's geometry descriptions. Using these as the basis for irregular device spaces could allow, for example, simulations to be carried out with different RHS modules used in each region.

5.6.6 Graphical User Interface

While the Beatbox scripting language presents a minimal, domain-specific means of specifying simulations, users without a background in computer programming may still find it intimidating. A graphical user interface capable of creating and editing scripts could greatly help bridge this gap. A useful interface for tools of this kind can be seen in Apple's Automator [Apple, Inc.].

Beatbox presents a powerful and flexible environment for cardiac simulation using high performance computing. By providing a minimal, intuitive, domain-specific language with which to specify simulations, Beatbox should cater to the needs of users from a broad range of backgrounds. By harnessing the power of distributed memory machines, Beatbox offers those users the scope to run large, complex simulations in a timely fashion.

APPENDIX

A

CODE LISTINGS FOR CHAPTER 2

```
#define CREATE_HEAD(name)\
int create_##name (Device *dev, char *w)  {\
    STR *S = (STR *) Malloc(sizeof(STR));\
    if (!S) ERROR1("cannot create %s",#name);
```

Listing A.1: *device.h* — The CREATE_HEAD template macro.

```
#define CREATE_TAIL(name,areaval)\
    dev->p = (Proc *)run_##name;\
    dev->d = (Proc *)destroy_##name;\
    dev->par  = S;\
    dev->w.area=areaval;\
    return(1);\
}
```

Listing A.2: *device.h* — The CREATE_TAIL template macro.

```
#define CONST(type,name) type name=S->name;
#define VAR(type,name)   type *name=&(S->name);
#define ARRAY(type,name) type *name=&(S->name[0]);
```

Listing A.3: *device.h* — Parameter access macros.

```
static int create (Name name, Create c, Device *d, char *rest) {

    // Assigns the variable often to d->c.
    if (!accept_condition(&(d->c),rest)) return 0;

    // Assigns x0=50, x1=99 to fields of d->s.
    // Others default to y0=Y0, y1=Y1, z0=Z0, z1=Z1, v0=0, v1=vmax-1.
    if (!accept_space(&(d->s),rest)) return 0;

    // Assigns default window parameters to d->w.
    if (!accept_window(&(d->w),rest)) return 0;

    // Assigns "foo" to d->n.
    if (!accepts("name=",&(d->n[0]),name,rest)) return 0;
    if (strcmp(d->n,name)) if(!def_dev(d)) return 0;

    return(c(d,rest));
}
```

Listing A.4: *init.c* — The create() function.

```
void term(void) {       /* free all dynamical objects */
  /*int idev;*/
  Device d;
  if (ndev) for (idev=ndev-1;idev>=0;idev--) {
    d=dev[idev];
    d.d(d.s,d.w,d.par);
  }
  state_free();         /* free the state array(s) */
  term_const();         /* clear the symbol table and free variables */
}
```

Listing A.5: *init.c* — The term() function.

```
ACCEPTI(rest,0,0,INONE);
if (S->rest) {
    if (S->u) MESSAGE("\n/* WARNING: finding resting state while already defined by the rhs */");
    else CALLOC(S->u,nv,sizeof(real));
    if (S->var.n) MESSAGE("\n/* WARNING: finding resting state while parameters are variable */");
    for(step=0;step<S->rest;step++) {
        if(!S->f(S->u,S->du,S->p,S->var,nv)) break;
        for(iv=0;iv<nv;iv++) (S->u)[iv]+=S->ht*S->du[iv];
    } /* for step */
} /* if S->rest */

if (S->u) {
    MESSAGE0("\n/* Resting state: "); for(iv=0;iv<nv;iv++) MESSAGE1("%lg ",(S->u)[iv]); MESSAGE0("*/");
    if (!already_initiated){
        for (ix=dev->s.x0;ix<=dev->s.x1;ix++){
            for (iy=dev->s.y0;iy<=dev->s.y1;iy++){
                for (iz=dev->s.z0;iz<=dev->s.z1;iz++){
                    for (iv=dev->s.v0;iv<=dev->s.v1;iv++){
                        New[ind(ix,iy,iz,iv)]=(S->u)[iv-dev->s.v0];
                    } /* for iv */
                } /* for iz */
            } /* for iy */
        } /* for ix */
    } /* if !already_initialised */
} else {
    CALLOC(S->u,nv,sizeof(real));
}
```

Listing A.6: *euler.c* — Setting initial conditions.

```
// Main loop through 3 dimensions
for(z=s.z0;z<=s.z1;z++) {
    for(y=s.y0;y<=s.y1;y++) {
        for(x=s.x0;x<=s.x1;x++) {

            // Point u to location of this cell in New
                u = (real *)(New + ind(x,y,z,s.v0));

            // Call the RhsProc, f()
                if(!f(u,du,p,var,s.v1-s.v0+1)) {

                ... // Report errors
                }
            // Update u using derivatives in du
                for(v=0;v<s.v1-s.v0+1;v++) u[v] += ht*du[v];
        }
    }
}
```

Listing A.7: *euler.c* — Main loop of `euler` assigning `u`, calling the cell model's `RhsProc` function and updating `u` using the derivatives in `du`.

```
// Count @ symbols in p. Store the result in var->n.
for(var->n=0;*p;var->n+=(*(p++)==AT));

// If there are dependent parameters, make space in Var.
    if(var->n){
        CALLOC(var->dst,var->n,sizeof(real *));
        CALLOC(var->src,var->n,sizeof(int));
    }

// If not, empty Var.
    else{var->src=NULL;(var->dst)=NULL;}
```

Listing A.8: *rhs.h* — Counting dependent parameters in the `RHS_CREATE_HEAD` template macro.

```
// Copy dependent parameters from u to S.
if (var.n) {
    for(ivar=0;ivar<var.n;ivar++) {
        *(var.dst[ivar])=u[var.src[ivar]];
    }
}
```

Listing A.9: *rhs.h* — Copying dependent parameters to `s` in the `RHS_HEAD` template macro.

```
real gam=D/(hx*hx);
for(z=s.z0;z<=s.z1;z++) {
  for(y=s.y0;y<=s.y1;y++) {
    for(x=s.x0;x<=s.x1;x++) {
      u=New+ind(x,y,z,s.v0);
      u[DV*(s.v1-s.v0)] = gam*(u[DX]+u[-DX]-2*u[0]);
    }
  }
}
```

Listing A.10: *diff1d.c* — Computing the laplacian in one dimension.

```
typedef struct {
    int ncode;
    pp_fn  *code; /*[ncode]*/
    p_real *data; /*[ncode]*/
    double  *u;   /*[vmax]*/
    real x, y, z;
    int np, nv;
    real *a;    /*[np*nv]*/
    double *p;  /*[nv]*/
    double phaseu, phasep;
    char filename[MAXPATH];
    FILE *file;
    char debugname[MAXPATH];
    FILE *debug;
    p_tb loctb;
} STR;
```

Listing A.11: *k_func.c* — The k_func device's parameter structure, after inclusion of *k_code.h*.

```
if(S->file) {
    /* ... */
    S->a = array;
    S->np = np;
    S->nv = nv;
    S->p  = pv;
} else {
    S->a=S->p=NULL;
    S->np=S->nv=0;
}
```

Listing A.12: *k_func.c* — Assigning values to fields of the parameter structure.

```
#undef SEPARATORS
#define SEPARATORS ";"
#define BLANK " \n\t\r"
BEGINBLOCK("pgm=",buf); {

  int idata, icode;
  char *pcode, *pdata;
  char *s1=strdup(buf);
  for(S->ncode=0;strtok(S->ncode?NULL:s1,SEPARATORS);S->ncode++);
  if (!S->ncode) ERROR1("no statements in \"%s\"",buf);
  FREE(s1);

  CALLOC(S->code,S->ncode,sizeof(pp_fn))
  CALLOC(S->data,S->ncode,sizeof(p_real))

  for(icode=0;icode<S->ncode;icode++) {
    for(pdata=strtok(icode?NULL:buf,SEPARATORS);*pdata&&strchr(BLANK,*pdata);pdata++);
    if(!*pdata) {S->code[icode]=NULL;continue;}
    for(pcode=pdata;(*pcode)&&(*pcode!='=');pcode++) if(strchr(BLANK,*pcode)) *pcode='\0';
    if(!*pcode) ERROR2(": no \'=\' sign in expression #%d: \'%s\'",icode+1,pdata);
    *(pcode++)='\0';
    if NOT(idata=tb_find(loctb,pdata))ERROR1("unknown symbol %s",pdata);
    /*if(tb_type(loctb,idata)!=t_real)  ERROR1("symbol %s is not double",pdata);*/
    if((tb_flag(loctb,idata)&f_vb)==0)  ERROR1("symbol %s is not a variable",pdata);
    S->data[icode] = (p_vb) tb_addr(loctb,idata);
    /*S->code[icode] = compile(pcode,loctb,t_real); CHK(pcode);*/
    S->code[icode] = compile(pcode,loctb,tb_type(loctb,idata)); CHK(pcode);
    MESSAGE3("\x01\n %s=%s%c",pdata,pcode,SEPARATORS[0]);
  }

} ENDBLOCK;
#undef SEPARATORS
#undef BLANK
```

Listing A.13: *k_comp.h* — Compiling the k_func program.

APPENDIX

B

QUI PREPROCESSOR MACROS

B.1 Integer

`ACCEPTI()` accepts an integer value using `accepti()`.

B.1.1 Usage

```
ACCEPTI(name,deflt,minv,maxv);
```

B.1.2 Parameters

`name` Name of the parameter whose value is to be read.

`deflt` Default value, in case no parameter called "name" is found. If INONE, a script parameter MUST be found.

`minv` Minimum allowed value. If INONE, no minimum is prescribed.

`maxv` Maximum allowed value. If INONE, no maximum is prescribed.

B.2 Long Integer

Accepts a long integer value using `acceptl()`.

B.2.1 Usage

```
ACCEPTL(name,deflt,minv,maxv);
```

B.2.2 Parameters

`name` Name of the parameter whose value is to be read.

`deflt` Default value, in case no parameter called "name" is found. If LNONE, a script parameter MUST be found.

`minv` Minimum allowed value. If LNONE, no minimum is prescribed.

`maxv` Maximum allowed value. If LNONE, no maximum is prescribed.

B.3 Real Number

`ACCEPTR()` accepts a real (double-precision floating point) value using `acceptr()`.

B.3.1 Usage

```
ACCEPTR(name,deflt,minv,maxv);
```

B.3.2 Parameters

`name` Name of the parameter whose value is to be read.

`deflt` Default value, in case no parameter called "name" is found. If RNONE, a script parameter MUST be found.

`minv` Minimum allowed value. If RNONE, no minimum is prescribed.

`maxv` Maximum allowed value. If RNONE, no maximum is prescribed.

B.4 String

`ACCEPTS()` accepts a string using `accepts()`.

B.4.1 Usage

```
ACCEPTS(name,deflt);
```

B.4.2 Parameters

`name` Name of the parameter whose value is to be read.

`deflt` Default value, in case no parameter called "name" is found. If NULL, a script parameter MUST be found.

B.5 File

`ACCEPTF()` accepts a file handle using `acceptf()`.

B.5.1 Usage

```
ACCEPTF(name,mode,default);
```

B.5.2 Parameters

`name` Name of the parameter whose value is to be read.

`mode` Access mode in which the file is to be opened. Syntax is identical to a fopen() call:

`r` **or** `rb` Open existing le for reading.

`w` **or** `wb` Create le or wipe existing le before writing.

a or ab Append to end of existing le, creating if necessary.

rt or rbt or rtb Open existing le for updatereading and writing.

wt or wbt or wtb Create le or wipe existing le before updating.

at or abt or atb Append — Open or create le for update, writing at end of le.

`deflt` Default filename, in case no parameter called "name" is found. If NULL, a filename MUST be found in the script.

In order to use `ACCEPTF`, a `char` array of size `MAXPATH` should be added to the parameter structure to hold the file's name, as opposed to `name`, which holds the file handle. This additional parameter should be named `name`+"name". For example, if `ACCEPTF()` is called as follows:

```
ACCEPTF("file","r",NULL);
```

the following fields will be required in the parameter structure:

```
char filename[MAXPATH]; // Relative path to file
FILE *file; // The file handle.
```

B.6 Code String

`ACCEPTC()` parses a code-string; an expression in QUI script syntax that may be evaluated using calc(). A code-string is fetched as a complete expression rather than as a reduced value so that it may be computed at runtime using k_variables. Uses `acceptc()`.

B.6.1 Usage

```
ACCEPTC(name,k_type,deflt);
```

B.6.2 Parameters

`name` Name of the parameter whose value is to be read.

`k_type` Data type returned by the code-string when executed.

`var` Address of variable where value is to be assigned.

`deflt` Default value, in case no parameter called "name" is found. If NULL, a script parameter MUST be found.

`codestring` Address at which the executable code should be stored.

`code` Pointer to the compiled code block.

`w` Parameter string to search.

B.7 Code Block

`BEGINBLOCK` accepts a block of QUI script code from a given parameter string. A block is any number of lines of code contained in braces (‘ ‘).

B.7.1 Usage

```
BEGINBLOCK(name,var);

ENDBLOCK;
```

B.7.2 Parameters

`name` Name of the script parameter to which the block is assigned, e.g. `LOCKname=B`.

`var` Address of the variable to which the code block is to be assigned.

`w` Parameter string to search.

APPENDIX

C

CODE LISTINGS FOR CHAPTER 3

```
void decomp_getSuperindices(int rank, int *ix, int *iy, int *iz){
    int rt;
    *ix = rank / (mpi_ny * mpi_nz);
    rt = rank % (mpi_ny * mpi_nz);
    *iy = rt / mpi_nz;
    *iz = rt % mpi_nz;
}
```

Listing C.1: *decomp.c* — Computing superindices.

```
/*  Approximate length of subdomain along each axis.    */
x_slice_size = (xmax-(ONE*2)) / mpi_nx;
y_slice_size = (ymax-(TWO*2)) / mpi_ny;
z_slice_size = (zmax-(TRI*2)) / mpi_nz;

/* Remainders from i[xyz] * [xyz]_slice_size */
/* The remainder will be shared amongst the first processes on the axis */
x_remainder  = (xmax-(ONE*2)) % mpi_nx;
y_remainder  = (ymax-(TWO*2)) % mpi_ny;
z_remainder  = (zmax-(TRI*2)) % mpi_nz;

for(rank=0;rank<num_active_procs;rank++){
    decomp_getSuperindices(rank,&ix,&iy,&iz);

    /*Lower x bound */
    subdomain_limits[rank].local_xmin   = ONE + (ix * x_slice_size) + (ix < x_remainder? ix :x_remainder);
    /*Upper x bound */
    subdomain_limits[rank].local_xmax   = ONE + ((ix+1) * x_slice_size) + (ix < x_remainder? (ix+1) :x_remainder);

    /*Lower y bound */
    subdomain_limits[rank].local_ymin   = TWO + (iy * y_slice_size) + (iy < y_remainder? iy :y_remainder);
    /*Upper y bound */
    subdomain_limits[rank].local_ymax   = TWO + ((iy+1) * y_slice_size) + (iy < y_remainder? (iy+1) :y_remainder);

    /*Lower z bound */
    subdomain_limits[rank].local_zmin   = TRI + (iz * z_slice_size) + (iz < z_remainder? iz :z_remainder);
    /*Upper z bound */
    subdomain_limits[rank].local_zmax   = TRI + ((iz+1) * z_slice_size) + (iz < z_remainder? (iz+1) :z_remainder);
} /* for(rank) */

/*  Assign own subdomain boundaries */
local_xmin = subdomain_limits[mpi_rank].local_xmin;
local_xmax = subdomain_limits[mpi_rank].local_xmax;
local_ymin = subdomain_limits[mpi_rank].local_ymin;
local_ymax = subdomain_limits[mpi_rank].local_ymax;
local_zmin = subdomain_limits[mpi_rank].local_zmin;
local_zmax = subdomain_limits[mpi_rank].local_zmax;
```

Listing C.2: *decomp.c* — Computing subdomain boundaries.

```
int haloSwap(){
/* X AXIS */
    if(mpi_ix % 2 == 0){
        if(mpi_ix < mpi_nx-1){
            mpi_errno = MPI_Sendrecv(New, 1, XP_Type, XP_NEIGHBOUR, 0, New, 1, XP_Halo_Type,
                                     XP_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }else{
        /*  No need to check if ix > 0, since ix is odd here. */
            mpi_errno = MPI_Sendrecv(New, 1, XN_Type, XN_NEIGHBOUR, 0, New, 1, XN_Halo_Type,
                                     XN_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
    }
```

```
    CHECK_MPI_SUCCESS("Failed to swap buffers.")
    if(mpi_ix % 2 == 1){
        if(mpi_ix < mpi_nx-1){
            mpi_errno = MPI_Sendrecv(New, 1, XP_Type, XP_NEIGHBOUR, 0, New, 1, XP_Halo_Type,
                                     XP_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }else{
        if(mpi_ix > 0){
            mpi_errno = MPI_Sendrecv(New, 1, XN_Type, XN_NEIGHBOUR, 0, New, 1, XN_Halo_Type,
                                     XN_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }
    CHECK_MPI_SUCCESS("Failed to swap buffers.")

/* Y AXIS */
    if(mpi_iy % 2 == 0){
        if(mpi_iy < mpi_ny-1){
            mpi_errno = MPI_Sendrecv(New, 1, YP_Type, YP_NEIGHBOUR, 0, New, 1, YP_Halo_Type,
                                     YP_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }else{
        if(mpi_iy > 0){
            mpi_errno = MPI_Sendrecv(New, 1, YN_Type, YN_NEIGHBOUR, 0, New, 1, YN_Halo_Type,
                                     YN_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }
    CHECK_MPI_SUCCESS("Failed to swap buffers.")
    if(mpi_iy % 2 == 1){
        if(mpi_iy < mpi_ny-1){
            mpi_errno = MPI_Sendrecv(New, 1, YP_Type, YP_NEIGHBOUR, 0, New, 1, YP_Halo_Type,
                                     YP_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }else{
        if(mpi_iy > 0){
            mpi_errno = MPI_Sendrecv(New, 1, YN_Type, YN_NEIGHBOUR, 0, New, 1, YN_Halo_Type,
                                     YN_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }
    CHECK_MPI_SUCCESS("Failed to swap buffers.")

/* Z AXIS */
    if(mpi_iz % 2 == 0){
        if(mpi_iz < mpi_nz-1){
            mpi_errno = MPI_Sendrecv(New, 1, ZP_Type, ZP_NEIGHBOUR, 0, New, 1, ZP_Halo_Type,
                                     ZP_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }else{
        if(mpi_iz > 0){
            mpi_errno = MPI_Sendrecv(New, 1, ZN_Type, ZN_NEIGHBOUR, 0, New, 1, ZN_Halo_Type,
                                     ZN_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }
    CHECK_MPI_SUCCESS("Failed to swap buffers.")
    if(mpi_iz % 2 == 1){
        if(mpi_iz < mpi_nz-1){
            mpi_errno = MPI_Sendrecv(New, 1, ZP_Type, ZP_NEIGHBOUR, 0, New, 1, ZP_Halo_Type,
                                     ZP_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }else{
        if(mpi_iz > 0){
            mpi_errno = MPI_Sendrecv(New, 1, ZN_Type, ZN_NEIGHBOUR, 0, New, 1, ZN_Halo_Type,
                                     ZN_NEIGHBOUR, 0, ALL_ACTIVE_PROCS, MPI_STATUS_IGNORE);
        }
    }
    CHECK_MPI_SUCCESS("Failed to swap buffers.")
    return 1;
}
```

Listing C.3: *device.c* — The `haloswap()` function.

```
void decomp_defineHaloTypes(){
    /*  Types for Halo Swapping
     *
     *  Labelled as XN, XP, YN, YP, ZN, ZP
     *  where, e.g. XN is the YZV hyperplane at local_xmin.
     *  ZP is the XYV hyperplane at local_zmax.
     *
     *  ZP_Halo_Type is the halo at local_xmax.
     *  YN_Halo_Type is the halo at local_ymin-1.
     *
     *  Hyperplanes exchanged in the y axis are widened in
     *  the x axis, and those exchanged in the z axis are widened in
     *  in the x and y axes. This includes points from diagonally
     *  neighbouring subdomains, which prevents the need for exchanging
     *  corner points explicitly between these processes.
     *  ---------------------------------- */
    #define NDIMS 4
    int sizes[NDIMS],subsizes[NDIMS],starts[NDIMS];
    sizes[0] = xlen;
    sizes[1] = ylen;
    sizes[2] = zlen;
    sizes[3] = vmax;

    if(mpi_nx > 1){
        /*  XN  */
        subsizes[0] = 1;
```

```
      subsizes[1] = local_ymax-local_ymin;
      subsizes[2] = local_zmax-local_zmin;
      subsizes[3] = vmax;

      starts[0] = ONE;
      starts[1] = TWO;
      starts[2] = TRI;
      starts[3] = 0;

      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &XN_Type);
      CHECK_MPI_SUCCESS("Couldn't define XN_Type.")
      mpi_errno = MPI_Type_commit(&XN_Type);
      CHECK_MPI_SUCCESS("Couldn't commit XN_Type.")

      /*  XN Halo */
      starts[0] -= 1;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &XN_Halo_Type);
      CHECK_MPI_SUCCESS("Couldn't define XN_Halo_Type.")
      mpi_errno = MPI_Type_commit(&XN_Halo_Type);
      CHECK_MPI_SUCCESS("Couldn't commit XN_Halo_Type.")

      /*  XP  */
      starts[0] = (local_xmax-1 + ONE) - local_xmin;
      starts[1] = TWO;
      starts[2] = TRI;
      starts[3] = 0;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &XP_Type);
      CHECK_MPI_SUCCESS("Couldn't define XP_Type.")
      mpi_errno = MPI_Type_commit(&XP_Type);
      CHECK_MPI_SUCCESS("Couldn't commit XP_Type.")

      /*  XP Halo */
      starts[0] += 1;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &XP_Halo_Type);
      CHECK_MPI_SUCCESS("Couldn't define XP_Halo_Type.")
      mpi_errno = MPI_Type_commit(&XP_Halo_Type);
      CHECK_MPI_SUCCESS("Couldn't commit XP_Halo_Type.")
  } /*    if(mpi_nx > 1)  */
  if(mpi_ny > 1){
      /*  YN  */
      subsizes[0] = xlen; /*  Widened to include magic corners.   */
      subsizes[1] = 1;
      subsizes[2] = local_zmax-local_zmin;
      subsizes[3] = vmax;

      starts[0] = 0;      /*  0 to include magic corner.  */
      starts[1] = TWO;
      starts[2] = TRI;
      starts[3] = 0;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &YN_Type);
      CHECK_MPI_SUCCESS("Couldn't define YN_Type.")
      mpi_errno = MPI_Type_commit(&YN_Type);
      CHECK_MPI_SUCCESS("Couldn't commit YN_Type.")

      /*  YN Halo     */
      starts[1] -= 1;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &YN_Halo_Type);
      CHECK_MPI_SUCCESS("Couldn't define YN_Halo_Type.")
      mpi_errno = MPI_Type_commit(&YN_Halo_Type);
      CHECK_MPI_SUCCESS("Couldn't commit YN_Halo_Type.")

      /*  YP  */
      starts[0] = 0;      /*  0 to include magic corner.  */
      starts[1] = (local_ymax-1 + TWO) - local_ymin;
      starts[2] = TRI;
      starts[3] = 0;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &YP_Type);
      CHECK_MPI_SUCCESS("Couldn't define YP_Type.")
      mpi_errno = MPI_Type_commit(&YP_Type);
      CHECK_MPI_SUCCESS("Couldn't commit YP_Type.")

      /*  YP Halo */
      starts[1] += 1;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &YP_Halo_Type);
      CHECK_MPI_SUCCESS("Couldn't define YP_Type.")
      mpi_errno = MPI_Type_commit(&YP_Halo_Type);
      CHECK_MPI_SUCCESS("Couldn't commit YP_Type.")
  } /*    if(mpi_ny > 1)  */
  if(mpi_nz > 1){
      /*  ZN  */
      subsizes[0] = xlen; /*  Widened to include magic corners    */
      subsizes[1] = ylen; /*  Widened to include magic corners    */
      subsizes[2] = 1;
      subsizes[3] = vmax;

      starts[0] = 0;      /*  0 to include magic corner.  */
      starts[1] = 0;      /*  0 to include magic corner.  */
      starts[2] = TRI;
      starts[3] = 0;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &ZN_Type);
      CHECK_MPI_SUCCESS("Couldn't define ZN_Type.")
      mpi_errno = MPI_Type_commit(&ZN_Type);
      CHECK_MPI_SUCCESS("Couldn't commit ZN_Type.")

      /*  ZN Halo */
      starts[2] -= 1;
      mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &ZN_Halo_Type);
```

```
        CHECK_MPI_SUCCESS("Couldn't define ZN_Halo_Type.")
        mpi_errno = MPI_Type_commit(&ZN_Halo_Type);
        CHECK_MPI_SUCCESS("Couldn't commit ZN_Halo_Type.")

        /*  ZP  */
        starts[0] = 0;      /*  0 to include magic corner.  */
        starts[1] = 0;      /*  0 to include magic corner.  */
        starts[2] = (local_zmax-1 + TRI) - local_zmin;
        starts[3] = 0;
        mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &ZP_Type);
        CHECK_MPI_SUCCESS("Couldn't define ZP_Type.")
        mpi_errno = MPI_Type_commit(&ZP_Type);
        CHECK_MPI_SUCCESS("Couldn't commit ZP_Type.")

        /*  ZP Halo */
        starts[2] += 1;
        mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &ZP_Halo_Type);
        CHECK_MPI_SUCCESS("Couldn't define ZP_Halo_Type.")
        mpi_errno = MPI_Type_commit(&ZP_Halo_Type);
        CHECK_MPI_SUCCESS("Couldn't commit ZP_Halo_Type.")
    } /*    if(mpi_nz > 1)  */

    #undef NDIMS
}
```

Listing C.4: *decomp.c* — The `decomp_defineHaloTypes()` function.

```
/*  Define new group and communicator for I/O,
 *  including only instances with runHere = true.   */
int deviceCommunicatorWithFirstRank(int runHere, MPI_Comm *new_comm, int *first){
    /*  Group of existing communicator (ALL_ACTIVE_PROCS). */
    MPI_Group old_group;
    /*  Group for new communicator */
    MPI_Group new_group;
    /*  runHere flag from every process. */
    int runHeres[num_active_procs];
    /*  Number of processes in new group. */
    int group_count = 0;
    /*  Handy integers. */
    int i, next, errcode, success;

    mpi_errno = MPI_Allgather(&runHere, 1, MPI_INT, &runHeres, 1, MPI_INT, ALL_ACTIVE_PROCS);
    CHECK_MPI_SUCCESS("Couldn't gather runHere flags.")

    /*  Count running instances. */
    for(i=0; i<num_active_procs; i++){
        if(runHeres[i]){
            group_count++;
        }
    }

    /*  Identify ranks of running instances to include in new group. */
    int group_ranks[group_count];
    next = 0;
    for(i=0; i<num_active_procs; i++){
        if(runHeres[i]){
            group_ranks[next++] = i;
        }
    }

    *first = group_ranks[0];

    mpi_errno = MPI_Comm_group(ALL_ACTIVE_PROCS, &old_group);
    CHECK_MPI_SUCCESS("Couldn't get old communicator's group.")

    errcode = MPI_Group_incl(old_group, group_count, group_ranks, &new_group);
    CHECK_MPI_SUCCESS("Couldn't make new group.")

    errcode = MPI_Comm_create(ALL_ACTIVE_PROCS, new_group, new_comm);
    CHECK_MPI_SUCCESS("Couldn't create new communicator.")

    return 1;
}

/*  Overloaded for the majority of occasions where first isn't necessary. */
int deviceCommunicator(int runHere, MPI_Comm *new_comm){
    int first;
    return deviceCommunicatorWithFirstRank(runHere, new_comm, &first);
}
```

Listing C.5: *device.c* — The `deviceCommunicatorWithFirstRank()` and `deviceCommunicator()` functions.

```
#define DEVICE_REQUIRES_SYNC dev->sync = 1;
```

Listing C.6: *device.h* — MPI version of the `DEVICE_REQUIRES_SYNC` macro.

```
#define DEVICE_ALWAYS_RUNS dev->alwaysRun = 1;
```

Listing C.7: *device.h* — MPI version of the `DEVICE_ALWAYS_RUNS` macro.

```
#define DEVICE_IS_SPACELESS                                                                                 \
if(spaceParametersExist(w)){                                                                                \
    MESSAGE("WARNING: %s does not use space parameters (x0,x1,y0,y1,z0,z1).\nThe parameters provided will be ignored.",dev->n); \
}
```

Listing C.8: *device.h* — The `DEVICE_IS_SPACELESS` macro.

```
#define DEVICE_HAS_DEFAULT_SPACE                                                                            \
if(spaceParametersExist(w)){                                                                                \
    EXPECTED_ERROR1("%s insists on using the default space.\n                                               \
                    Please try again without space parameters (x0,x1,y0,y1,z0,z1).\n",dev->n);  \
}
```

Listing C.9: *device.h* — The `DEVICE_HAS_DEFAULT_SPACE` macro.

```
#define DEVICE_MUST_BE_NOWHERE                                                                              \
if(dev->s.nowhere != 1){                                                                                    \
    ERROR1("%s requires that nowhere is equal to 1. Try again, adding 'nowhere=1' to the device parameters.\n",dev->n); \
}
```

Listing C.10: *device.h* — The `DEVICE_MUST_BE_NOWHERE` macro.

```
#define DEVICE_OPERATES_ON_A_SINGLE_POINT                                                   \
if( (find_key("x1=",w)) ||                                                                  \
    (find_key("y1=",w)) ||                                                                  \
    (find_key("z1=",w)) ||                                                                  \
    (find_key("v1=",w))                                                                     \
){                                                                                          \
    MESSAGE("WARNING: %S does not use the x1,y1,z1 or v1 space parameters",dev->n);         \
    MESSAGE(" -- it only operates on a single point.");                                     \
    MESSAGE(" The parameter(s) provided will be ignored.\n");                               \
}
```

Listing C.11: *device.h* — The `DEVICE_OPERATES_ON_A_SINGLE_POINT` macro.

```
DEVICE_CONST(real,D)
real gam=D/(hx*hx);
for(x=s.x0;x<=s.x1;x++) {
    for(y=s.y0;y<=s.y1;y++){
        for(z=s.z0;z<=s.z1;z++){
            if(isTissue(x,y,z)){
                sum = 0;
                    u=New+ind(x,y,z,s.v0);
                value = u[0];
                if(dim>=1){
                    sum += isTissue(x-1,y  ,z  )? u[-DX] : value;
                    sum += isTissue(x+1,y  ,z  )? u[ DX] : value;
                }
                if(dim>=2){
                    sum += isTissue(x  ,y-1,z  )? u[-DY] : value;
                    sum += isTissue(x  ,y+1,z  )? u[ DY] : value;
                }
                if(dim>=3){
                    sum += isTissue(x  ,y  ,z-1)? u[-DZ] : value;
                    sum += isTissue(x  ,y  ,z+1)? u[ DZ] : value;
                }
                u[DV*(s.v1-s.v0)] = gam * ( sum - (2*dim*u[0]) );
            } /* if isTissue() */
        } /* for z */
    } /* for y */
} /* for x */
```

Listing C.12: *diff.c* — Consolidated computation of the Laplacian and Neumann boundary conditions in the `diff` device's run function.

```
int accept_real_variable(real **v, char *name, char *w) {
    typedef struct {char var[80];} STR;
    STR s;
    STR *S=&s;
    int i;
    char key[80];
    sprintf(key,"%s=",name);
    accepts(key,&(S->var[0]),"",w);
    if(!strcmp(S->var,"")){
        ERROR2("ERROR: cannot find word \"%s\" in the string:\n\"%s\"\n",name,w);
    }
    k_on();
    if NOT(i=tb_find(deftb,S->var)) ERROR1("unknown symbol %s",S->var);
    if (tb_type(deftb,i)!=t_real) ERROR1("symbol %s is not double",S->var);
    if ((tb_flag(deftb,i)&f_ro)==f_ro) ERROR("symbol '%s' is read-only",S->var);
    if ((tb_flag(deftb,i)&f_vb)==0) ERROR1("symbol %s is not a variable",S->var);
    *v = (double *)tb_addr(deftb,i);
    return 1;
}
```

Listing C.13: *qpp.c* — The `accept_real_variable()` function. The specified variable must be defined in `deftb`, of type `real` and not read-only.

```
#define ACCEPTV(a)   if (!accept_real_variable(&(S->a),#a,w)) return(0)
```

Listing C.14: *qpp.h* — The `ACCEPTV()` function macro.

```
CREATE_HEAD(sample)
    DEVICE_ALWAYS_RUNS
    DEVICE_OPERATES_ON_A_SINGLE_POINT
    ACCEPTV(result);
#if MPI
    S->root = getRankContainingPoint(dev->s.global_x0,dev->s.global_y0,dev->s.global_z0);
    /* Only root process writes to debug */
    if(mpi_rank == S->root){
#endif
        ACCEPTF(debug,"wt","");
#if MPI
    }
#endif
CREATE_TAIL(sample,1)
```

Listing C.15: *sample.c* — The `sample` device's Create function.

```
RUN_HEAD(sample)
#if MPI
    DEVICE_CONST(int,root);
#endif
    DEVICE_CONST(real *,result);
#if MPI
    if(mpi_rank == root){
#endif
        DEVICE_CONST(FILE *,debug)
        *result = New[ind(s.x0,s.y0,s.z0,s.v0)];
        if (debug) fprintf(debug, "%ld : %lf\n",t,*result);
#if MPI
    }/* if root */
    MPI_Bcast(S->result, 1, MPI_DOUBLE, root, ALL_ACTIVE_PROCS);
#endif
RUN_TAIL(sample)
```

Listing C.16: *sample.c* — The `sample` device's Run function.

```
CREATE_HEAD(reduce)
    DEVICE_ALWAYS_RUNS
    ACCEPTS(operation,"");
    if(strcmp(S->operation, "min")==0){
        S->local_reduce = reduce_min;
#if MPI
        S->global_reduce = MPI_MIN;
#endif
    }else if(strcmp(S->operation, "max")==0){
        S->local_reduce = reduce_max;
#if MPI
        S->global_reduce = MPI_MAX;
#endif
    }else if(strcmp(S->operation, "sum")==0){
        S->local_reduce = reduce_sum;
#if MPI
        S->global_reduce = MPI_SUM;
#endif
    }else if(strcmp(S->operation, "prod")==0){
        S->local_reduce = reduce_prod;
#if MPI
        S->global_reduce = MPI_PROD;
#endif
    }else{
        ERROR("The given operation, \"%s\", was not recognised.\n
               Accepted values are \"min\", \"max\", \"sum\" and \"prod\".\n",S->operation);
    }
    ACCEPTV(result);
#if MPI
    int device_rank, i_am_root, root, i;
    int rootFlags[num_active_procs];
    deviceCommunicator(dev->s.runHere, &(S->comm));

    /*  MPI_Reduce operates on the device communicator, defined where s.runHere is true.
     *  As the MPI_Broadcast that distributes the result must operate on ALL_ACTIVE_PROCS,
     *  it's necessary to identify the rank of the device root in ALL_ACTIVE_PROCS.
     *
     *  The MPI_Reduce root is 0 in the device communicator.
     *  The MPI_Broadcast root is computed here and stored in S->root.
     */

    i_am_root = 0;
    if(dev->s.runHere){
        MPI_Comm_rank(S->comm,&device_rank);
        if(device_rank == 0) i_am_root = 1;
    }
    mpi_errno = MPI_Allgather(&i_am_root, 1, MPI_INT, &rootFlags, 1, MPI_INT, ALL_ACTIVE_PROCS);
    CHECK_MPI_SUCCESS("Couldn't gather root flags.")
    for(i=0; i<num_active_procs; i++){
        if(rootFlags[i]) root = i;
    }
```

```
    S->root = root;

#endif
CREATE_TAIL(reduce,1)
```

Listing C.17: *reduce.c* — The `reduce` device's Create function.

```
RUN_HEAD(reduce)
    double running = New[ind(s.x0,s.y0,s.z0,s.v0)];
    DEVICE_VAR(Reduce *, local_reduce)

#if MPI
    DEVICE_CONST(MPI_Comm, comm)
    DEVICE_CONST(int, root)
    DEVICE_CONST(MPI_Op, global_reduce)

    // Only active processes check their local max.
    if(s.runHere){
#endif
        int x, y, z, v;
        int firstPass = 1;
        double this_value;
        for(x=s.x0;x<=s.x1;x++){
            for(y=s.y0;y<=s.y1;y++){
                for(z=s.z0;z<=s.z1;z++){
                    for(v=s.v0;v<=s.v1;v++){
                        this_value = New[ind(x,y,z,v)];
                        if(firstPass){
                            running = this_value;
                            firstPass = 0;
                        }else{
                            (*local_reduce)(this_value, &running);
                        }
                    } // for v
                } // for z
            } // for y
        } // for x
#if MPI
        mpi_errno = MPI_Reduce(&running, S->result, 1, MPI_DOUBLE, global_reduce, 0, comm);
        CHECK_MPI_SUCCESS("Couldn't carry out reduction operation.")
    } // if(s.runHere)

    /* ------------- DISTRIBUTE RESULT TO ALL PROCESSES ---------------*/
    MPI_Bcast(S->result, 1, MPI_DOUBLE, root, ALL_ACTIVE_PROCS);
#else
    *(S->result) = running;
#endif
RUN_TAIL(reduce)
```

Listing C.18: *reduce.c* — The `reduce` device's Run function.

```
CREATE_HEAD(poincare)
    DEVICE_ALWAYS_RUNS
    DEVICE_OPERATES_ON_A_SINGLE_POINT
    ACCEPTV(result);
    S->timestep = NULL;
    if(find_key("timestep=",w)){
        ACCEPTV(timestep);
    }
    ACCEPTR(cross,RNONE,RNONE,RNONE);
    ACCEPTI(sign,BOTH,NEGATIVE,POSITIVE);
    S->first = 1;
#if MPI
    S->root = getRankContainingPoint(dev->s.global_x0,dev->s.global_y0,dev->s.global_z0);
#endif
CREATE_TAIL(poincare,1)
```

Listing C.19: *poincare.c* — The `poincare` device's Create function.

```
#include "argsused.h"
#include "assused.h"
RUN_HEAD(poincare)
#if MPI
    DEVICE_CONST(int,root);
    real buffer[2];
#endif
    DEVICE_CONST(real,then);
    DEVICE_CONST(real,cross);
    DEVICE_CONST(real *,result);
    DEVICE_CONST(real *,timestep);
    DEVICE_CONST(int,sign);
    int crossingDetected = 0;
    real now = 0;
#if MPI
    if(mpi_rank == root){
#endif
        now = New[ind(s.x0,s.y0,s.z0,s.v0)];    /* Current value */
#if MPI
    } /* if root */
#endif
    if(S->first){
```

```
        S->first = 0;
    }else{
#if MPI
        if(mpi_rank == root){
#endif
            /* Detect Crossing */
            if( (sign != POSITIVE && (then > cross && cross > now)) || /* Negative crossing */
                (sign != NEGATIVE && (then < cross && cross < now)) ){ /* Positive crossing */
                crossingDetected = 1;
                printf("Crossing detected at t=%ld.\n",t);
                if(timestep!=NULL) *timestep=t;
            }
            *result = crossingDetected;

#if MPI
        }/* if root */
        /* ------------ DISTRIBUTE RESULT TO ALL PROCESSES ---------------*/
        buffer[0] = *result;
        if(timestep) buffer[1] = *timestep;
        MPI_Bcast(&buffer, timestep?2:1, MPI_DOUBLE, root, ALL_ACTIVE_PROCS);
        *result = buffer[0];
        if(timestep) *timestep = buffer[1];
#endif
    }/* first=0 */
    S->then = now;
RUN_TAIL(poincare)
```

Listing C.20: *poincare.c* — The `poincare` device's Run function.

```
#define NDIMS 4
MPI_Datatype filetype;
MPI_Datatype sourceType;
int sizes[NDIMS],subsizes[NDIMS],starts[NDIMS];
/*-------------------------------------------------------------------------
 * This first datatype describes how the local space
 * contributes to the file (the global space)
 *-------------------------------------------------------------------------*/
/*  Global space dimensions. */
int xsize = (dev->s.global_x1 - dev->s.global_x0) + 1;
int ysize = (dev->s.global_y1 - dev->s.global_y0) + 1;
int zsize = (dev->s.global_z1 - dev->s.global_z0) + 1;
int vsize = (dev->s.v1 - dev->s.v0) + 1;

/*  Local space dimensions */
int local_xsize = (dev->s.x1 - dev->s.x0) + 1;
int local_ysize = (dev->s.y1 - dev->s.y0) + 1;
int local_zsize = (dev->s.z1 - dev->s.z0) + 1;
int local_vsize = vsize;

/*  Dimensions of the global space. */
sizes[0] = xsize;
sizes[1] = ysize;
sizes[2] = zsize;
sizes[3] = vsize;

/*  Dimensions of the local space. */
subsizes[0] = local_xsize;
subsizes[1] = local_ysize;
subsizes[2] = local_zsize;
subsizes[3] = local_vsize;

starts[0] = dev->s.x0 - dev->s.global_x0;
starts[1] = dev->s.y0 - dev->s.global_y0;
starts[2] = dev->s.z0 - dev->s.global_z0;
starts[3] = 0;

mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &filetype);
CHECK_MPI_SUCCESS("Couldn't define filetype.")
mpi_errno = MPI_Type_commit(&filetype);
CHECK_MPI_SUCCESS("Couldn't commit filetype.")

/*-------------------------------------------------------------------------
 * The datatype defined below describes how which data will
 * be pulled from New to be written to file.
 *
 * With this defined, it's possible to write a single block
 * of data to file containing the whole space.
 *-------------------------------------------------------------------------*/

/*  Dimensions of New. */
sizes[0] = (local_xmax-local_xmin)+(ONE*2);
sizes[1] = (local_ymax-local_ymin)+(TWO*2);
sizes[2] = (local_zmax-local_zmin)+(TRI*2);
sizes[3] = vmax;

/*  Position of local space in New. */
starts[0] = (dev->s.x0 + ONE) - local_xmin;
starts[1] = (dev->s.y0 + TWO) - local_ymin;
starts[2] = (dev->s.z0 + TRI) - local_zmin;
starts[3] = dev->s.v0;

mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &sourceType);
CHECK_MPI_SUCCESS("Couldn't define sourceType.")
mpi_errno = MPI_Type_commit(&sourceType);
CHECK_MPI_SUCCESS("Couldn't commit sourceType.")
```

```
#undef NDIMS
/*-----------------------------------------------------------------------------*/
```

Listing C.21: *dump_types.h* — Definitions of the derived datatypes used by the `dump`, `load` and `ctlpoint` devices.

```
CREATE_HEAD(dump)
    ACCEPTI(append,0,0,1);

#if MPI /*  Prepare for collective writing. */
    if (!accepts("file=",&(S->filename[0]),NULL,w)) return(0);

    /*  Communicator for active dump instances. */
    MPI_Comm comm_dump;

    /*  File */
    MPI_File file;
    int mode;
    int displacement=0;
    MPI_Offset filesize; /*  existing file size, for appending. */
    MPI_Aint prealloc_size;

    if(!deviceCommunicator(dev->s.runHere, &comm_dump)){
        ERROR("Could not create communicator.");
    }

    if(dev->s.runHere){ /*  Active devices only!     */

        /*  Only define types on active instances,
         *  as inactive instances will give invalid
         *  subarray dimensions. */
        #include "dump_types.h"

        /*  Set access mode */
        if(S->append){
            mode = MPI_MODE_WRONLY | MPI_MODE_CREATE | MPI_MODE_APPEND;
        }else{
            mode = MPI_MODE_WRONLY | MPI_MODE_CREATE;
        }

        /*  Open file and set view */
        mpi_errno = MPI_File_open(comm_dump, S->filename, mode, MPI_INFO_NULL, &file);

        /*  If requested by user, append to EOF */
        if(S->append){
            MPI_File_get_size(file, &filesize);
            displacement = (int)filesize;
        }

        mpi_errno = MPI_File_set_view(file, displacement, MPI_DOUBLE, filetype, "native", MPI_INFO_NULL);

        MPI_Type_extent(filetype,&prealloc_size);
        prealloc_size += displacement;
        MPI_File_preallocate(file,prealloc_size);

        /*  Assign the file to the parameter structure */
        S->file = file;
        S->sourceType = sourceType;
    }

#else /*  Sequential version */
    ACCEPTF(file,S->append?"ab":"wb",NULL);
#endif

CREATE_TAIL(dump,0)
```

Listing C.22: *dump.c* — The `dump` device's Create function.

```
RUN_HEAD(dump)
int x, y, z, v; /*  Loop indices */

#if MPI /*  MPI version */
DEVICE_CONST(int, append)
DEVICE_CONST(MPI_File, file)
DEVICE_CONST(MPI_Datatype, sourceType)

/*  Rewind file if not appending */
if(!append) MPI_File_seek(file,0,MPI_SEEK_SET);

/*  Write to file. */
mpi_errno = MPI_File_write(file, New, 1, sourceType, MPI_STATUS_IGNORE);
CHECK_MPI_SUCCESS("Couldn't write to file.")

#else /*  Sequential Version */
DEVICE_CONST(FILE *,file)
DEVICE_CONST(int, append)
if (!file) return 1;
if (!append) rewind(file);
if (s.x0==0 && s.x1==xmax-1 && s.y0==0 && s.y1==ymax-1 && s.z0==0 && s.z1==zmax-1 && s.v0==0 && s.v1==vmax-1){
    fwrite(New,sizeof(real),xmax*ymax*zmax*vmax,file);
}else{
    for(x=s.x0;x<=s.x1;x++){
        for(y=s.y0;y<=s.y1;y++){
```

```
            for(z=s.z0;z<=s.z1;z++){
                for(v=s.v0;v<=s.v1;v++){
                    fwrite(New+ind(x,y,z,v),sizeof(real),1,file);
                }
            }
        }
    }
}
fflush(file);
#endif
RUN_TAIL(dump)
```

Listing C.23: *dump.c* — The `dump` device's Run function.

```
RUN_HEAD(load)
#if MPI
    DEVICE_CONST(MPI_File, file)
    DEVICE_CONST(MPI_Datatype, sourceType)
    DEVICE_CONST(int, rewind)
    if (!file) return 1;

    /*  Can I get a rewind? */
    if(rewind) MPI_File_seek(file,0,MPI_SEEK_SET);

    mpi_errno = MPI_File_read(file, New, 1, sourceType, MPI_STATUS_IGNORE);
    CHECK_MPI_SUCCESS("Couldn't read from file.")

#else /*  Sequential Version */
    DEVICE_CONST(FILE *,file)
    DEVICE_CONST(int, rewind)
    int x, y, z, v;
    if (!file) return 1;
    if (rewind) Rewind(file);
    if (s.x0==0 && s.x1==xmax-1 && s.y0==0 && s.y1==ymax-1 && s.z0==0 && s.z1==zmax-1 && s.v0==0 && s.v1==vmax-1){
        fread(New,sizeof(real),xmax*ymax*zmax*vmax,file);
    }else{
        for(x=s.x0;x<=s.x1;x++){
            for(y=s.y0;y<=s.y1;y++){
                for(z=s.z0;z<=s.z1;z++){
                    for(v=s.v0;v<=s.v1;v++){
                        fread(New+ind(x,y,z,v),sizeof(real),1,file);
                    }
                }
            }
        }
    }
#endif
RUN_TAIL(load)
```

Listing C.24: *load.c* — The `load` device's run function.

```
CREATE_HEAD(ctlpoint)
    DEVICE_IS_SPACELESS

    ACCEPTS(file,NULL);
    ACCEPTI(enrich,0,0,1);
    #if RKV
      ACCEPTI(rkv,0,0,1);
    #endif
    ACCEPTF(debug,"wt","");
    S->last=LNONE;

#if MPI

    MPI_Datatype k_var_Type;
    k_var var;

    int i;
    int lengths[3] = {maxname, 1, 256};
    MPI_Aint *displacements = calloc(3, sizeof(MPI_Aint));
    MPI_Datatype old_datatypes[3] = {MPI_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR};

    MPI_Get_address(var.nm,     &displacements[0]);
    MPI_Get_address(&var.tp,    &displacements[1]);
    MPI_Get_address(var.val,    &displacements[2]);

    for(i = 2; i>=0; i--){
        displacements[i] -= displacements[0];
    }

    MPI_Type_create_struct(3, lengths, displacements, old_datatypes, &k_var_Type);
    MPI_Type_commit(&k_var_Type);

    /*  Global space must be whole medium (inc absolute boundaries.) */
    dev->s.runHere = 1; /*  Must run on all processes. */
    dev->s.global_x0 = 0;
    dev->s.global_x1 = xmax-1;
    dev->s.global_y0 = 0;
    dev->s.global_y1 = ymax-1;
    dev->s.global_z0 = 0;
    dev->s.global_z1 = zmax-1;
    dev->s.v0 = 0;
    dev->s.v1 = vmax-1;
```

```
    if(mpi_ix == 0){
        dev->s.x0 = dev->s.global_x0;
    }else{
        dev->s.x0 = local_xmin;
    }
    if(mpi_ix == mpi_nx-1){
        dev->s.x1 = dev->s.global_x1;
    }else{
        dev->s.x1 = local_xmax-1;
    }
    if(mpi_iy == 0){
        dev->s.y0 = dev->s.global_y0;
    }else{
        dev->s.y0 = local_ymin;
    }
    if(mpi_iy == mpi_nx-1){
        dev->s.y1 = dev->s.global_y1;
    }else{
        dev->s.y1 = local_ymax-1;
    }
    if(mpi_iz == 0){
        dev->s.z0 = dev->s.global_z0;
    }else{
        dev->s.z0 = local_zmin;
    }
    if(mpi_iz == mpi_nz-1){
        dev->s.z1 = dev->s.global_z1;
    }else{
        dev->s.z1 = local_zmax-1;
    }

    /*  Define types for New. */
    #include "dump_types.h"
    MPI_Aint filetype_extent;
    MPI_Type_extent(filetype, &filetype_extent);
    S->k_var_Type = k_var_Type;
    S->filetype = filetype;
    S->sourceType = sourceType;
    S->filetype_extent = filetype_extent;
#endif MPI
CREATE_TAIL(ctlpoint,0)
```

Listing C.25: *ctlpoint.c* — The `ctlpoint` device's Create function.

```
RUN_HEAD(ctlpoint)
    DEVICE_ARRAY(char,file)
    DEVICE_CONST(int,enrich)
#if RKV
    DEVICE_CONST(int,rkv)
#endif
    DEVICE_VAR(long,last)
    DEVICE_CONST(FILE *,debug)
#define DB if(debug) {fprintf(debug,
#define BD ); fflush(debug);}

    DEVICE_CONST(MPI_Datatype, k_var_Type)
    DEVICE_CONST(MPI_Datatype, filetype)
    DEVICE_CONST(MPI_Datatype, sourceType)
    DEVICE_CONST(MPI_Aint, filetype_extent)
    MPI_File f;
    MPI_Offset offset;
    int x,y,z,v;
    int mode;
    int file_exists;
    FILE *temp_file;

    char V_[32], *p;
    INT xm_,ym_,zm_,vm_;
    int i;
    k_var var;

    DB "\nctlpoint called at %ld\n",t BD

    if(*last==LNONE) {          /* never saved, try to restore */
        DB "\nreading\n" BD

        #define D(name,new,old,type,size,erraction)                                          \
        if(mpi_errno != MPI_SUCCESS){                                                        \
            MPI_Error_string(mpi_errno, error_string, &error_length);                        \
            MESSAGE("Process %d: error reading from %s at t=%d --- %s", mpi_rank, file, t, error_string);   \
            fflush(stdout);                                                                  \
            /*erraction;*/                                                                  \
        } else if (new!=old && memcmp(new,old,sizeof(type)*size)) {                          \
            MESSAGE("P%02d: wrong %s read from %s: %d(x%0x) instead of %d(x%0x) \n",         \
            mpi_rank,name,file,*(long *)new,*(long *)new,*(long *)old,*(long *)old);         \
            /*erraction;*/                                                                  \
        } else if (debug)                                                                    \
            fprintf(debug,"\nread %s[0]=%f\n",name,(float)(*(type *)(new)));

        #define CHAR_VAR(name, new, old, size, erraction)                                    \
        mpi_errno = MPI_File_read(f, new, size, MPI_CHAR, MPI_STATUS_IGNORE);                \
        D(name, new, old, char, size, erraction)

        #define INT_VAR(name, new, old, size, erraction)                                     \
        mpi_errno = MPI_File_read(f, new, size, MPI_LONG, MPI_STATUS_IGNORE);                \
        D(name, new, old, INT, size, erraction)
```

```
    #define K_VAR(name, new, old, size, erraction)                                                  \
    mpi_errno = MPI_File_read(f, new, size, k_var_Type, MPI_STATUS_IGNORE);                         \
    D(name, new, old, k_var, size, erraction)

    /*  Use sequential method to check file existance.
     *  Prevents MPI errors all over the shop.  */
    if NOT(temp_file=fopen(file,"rb")) {
        file_exists = 0;
        MESSAGE("Cannot read control point %s at t=%d.\n", file, t);
    }else{
        file_exists = 1;
        fclose(temp_file);
    }

    if(file_exists==1){
        /*  READ FROM DUMP FILE */
        mpi_errno = MPI_File_open(ALL_ACTIVE_PROCS, file, MPI_MODE_RDONLY, MPI_INFO_NULL, &f);
        mpi_errno = MPI_File_set_view(f, 0, MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);
        CHECK_MPI_SUCCESS("Couldn't set view.")
        #if RKV
        {
            MESSAGE("WARNING: RKV functionality is currently disabled in MPI mode.\n");
        }
        #endif /*  RKV */

        /*  Read first block of global data. */
        #include "ctlpoint.h"

        #undef CHAR_VAR
        #undef INT_VAR
        #undef K_VAR

        MPI_File_get_position(f, &offset);
        mpi_errno = MPI_File_set_view(f, offset, sourceType, filetype, "native", MPI_INFO_NULL);
        mpi_errno = MPI_File_read(f, New, 1, sourceType, MPI_STATUS_IGNORE);
        CHECK_MPI_SUCCESS("Couldn't read from file.")
        haloSwap();

        offset += filetype_extent;
        mpi_errno = MPI_File_set_view(f, offset, MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);

        for(;;) {
            mpi_errno = MPI_File_read(f, &var, 1, k_var_Type, MPI_STATUS_IGNORE);
            if(mpi_errno != MPI_SUCCESS){
                MPI_Error_string(mpi_errno, error_string, &error_length);
                printf("Process %d: error reading k_var from %s at t=%ld--- %s\n", mpi_rank, file, t, error_string);
                fflush(stdout);
                break;
            }

            if NOT(var.nm[0]) break;        /* eof mark */
            if NOT(i=tb_find(deftb,var.nm)) {
                if (!enrich) continue;
                if NOT(i=tb_insert_abstract(
                deftb,var.nm,var.tp,(p_vd)Calloc(1,sizetable[var.tp]),0,f_vb|f_rs
                )) ERROR1("cannot define variable %s",var.nm);
            } /*  if not defined. */
            if (tb_type(deftb,i)!=var.tp) ERROR1("variable %s of wrong type",var.nm);
            if ((tb_flag(deftb,i)&f_vb)==0) ERROR1("%s is not a variable",var.nm);
            memcpy(tb_addr(deftb,i),&(var.val),sizetable[var.tp]);
        } /*  for */
        mpi_errno = MPI_File_close(&f);
        CHECK_MPI_SUCCESS("Couldn't close file after reading.")
        f = MPI_FILE_NULL;
        #undef D
        DB "\nread ok\n" BD
    } /*  if(file_exists==1) */
} /*  if(*last==LNONE) */
else {              /* saved, try to save */
    DB "\nwriting\n" BD
    vm_=vmax;

    #define D(name,new,old,type,size,erraction)                                                     \
    if(mpi_errno != MPI_SUCCESS){                                                                   \
        MPI_Error_string(mpi_errno, error_string, &error_length);                   \
        MESSAGE("Process %d: error writing %s to %s at t=%ld --- %s\n",mpi_rank,name,file,t, error_string);\
        fflush(stdout);                                                                             \
        ERROR0("\n");                                                                               \
    }

    #define CHAR_VAR(name, new, old, size, erraction)                                               \
    mpi_errno = MPI_File_write(f, old, size, MPI_CHAR, MPI_STATUS_IGNORE);                                  \
    D(name,new,old,char,size,erraction)

    #define INT_VAR(name, new, old, size, erraction)                                                \
    mpi_errno = MPI_File_write(f, old, size, MPI_LONG, MPI_STATUS_IGNORE);                                  \
    D(name,new,old,INT,size,erraction)

    #define K_VAR(name, new, old, size, erraction)                                                  \
    mpi_errno = MPI_File_write(f, &var, size, k_var_Type, MPI_STATUS_IGNORE);                                   \
    D(name,new,old,k_var,size,erraction)

    strcpy(buf,file);
    for(p=buf+strlen(buf)-1;p>=buf&&*p=='~';p--);
    if(p<buf) {MESSAGE("cannot generate tempname for %s",file);return(0);}
    *p='~';
```

```
        mode = MPI_MODE_WRONLY | MPI_MODE_CREATE;
        if(MPI_File_open(ALL_ACTIVE_PROCS, buf, mode, MPI_INFO_NULL, &f) != MPI_SUCCESS){
            MESSAGE("System error %d: %s\n",mpi_errno,strerror(mpi_errno));
            ERROR1("cannot open dump file for writing %s", buf);
            perror(0);
            return(0);
        }

        /*  Set view with no displacement (top of file). */
        mpi_errno = MPI_File_set_view(f, 0, MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);
        CHECK_MPI_SUCCESS("Couldn't set view.")

        if(mpi_rank == 0){
            /*  Write first block of global data. */
            #include "ctlpoint.h"
        }

        MPI_File_get_position(f, &offset);
        MPI_Bcast(&offset, 1, MPI_INT, 0, ALL_ACTIVE_PROCS);
        mpi_errno = MPI_File_set_view(f, offset, sourceType, filetype, "native", MPI_INFO_NULL);
        mpi_errno = MPI_File_write(f, New, 1, sourceType, MPI_STATUS_IGNORE);
        CHECK_MPI_SUCCESS("Couldn't write to file.")


        /*  Adjust file pointer to after New. */
        offset += filetype_extent;
        mpi_errno = MPI_File_set_view(f, offset, MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);

        if(mpi_rank == 0){
            /*  Write more global data... */
            for(i=1;i<=maxtab;i++) {
                if(!*tb_name(deftb,i)) continue;
                if((tb_flag(deftb,i)&f_vb)==0) continue;
                memcpy(var.nm,tb_name(deftb,i),maxname);
                var.tp=tb_type(deftb,i);
                memcpy(&(var.val),tb_addr(deftb,i),sizetable[var.tp]);
                K_VAR(var.nm,&var,&var,1,return 0);
            } /*  for i-->maxtab */
            K_VAR("empty_var",&empty_var,&empty_var,1,return 0); /* eof mark */
        } /*  if(mpi_rank == 0) */

        /*
            TODO Supress error message when closing the file.
        */
        mpi_errno = MPI_File_close(&f);
        CHECK_MPI_SUCCESS("Couldn't close ctlpoint file.")
        f = MPI_FILE_NULL;

        /*
            TODO Figure out what's going on here.
        */
        #ifdef NOBLE
            rename(buf,file);
        #else
            if (dim<3) frename(buf,file);
            else        rename(buf,file);
        #endif

        sprintf(buf,"dumped at %ld to %s\n",t,file);
        /* if(!MESSAGE(buf)) crt_text(buf,w.row0,w.col0,15); */
        #undef D

        DB "\nwritten ok\n" BD

        #undef CHAR_VAR
        #undef INT_VAR
        #undef K_VAR
    }

    *last = t;
RUN_TAIL(ctlpoint)
```

Listing C.26: *ctlpoint.c* — The MPI branch of the `ctlpoint` device's Run function.

```
#if MPI
/* MPI version includes runHere so that a new communicator can be created. */
int acceptq (const char *name,const char *mode,const char *deflt,int runHere,sequence *q, char *w);
#else
int acceptq (const char *name,const char *mode,const char *deflt,sequence *q, char *w);
#endif
```

Listing C.27: *sequence.h* — Parallel and sequential versions of the `acceptq()` function definition.

```
#if MPI
#define ACCEPTQ(b,c,d)   if (!acceptq(#b"=",c,d,dev->s.runHere,&(S->b),w)) return(0)
#else
#define ACCEPTQ(b,c,d)   if (!acceptq(#b"=",c,d,&(S->b),w)) return(0)
#endif
```

Listing C.28: *sequence.h* — Parallel and sequential versions of the `ACCEPTQ()` function macro.

```
#if MPI
/*  Returns an MPI file access mode roughly  */
/*  equivalent to the given fopen() mode. */
int getMPIFileMode(char *fm){
    int MPImode;

    if (        0==strcmp(fm,"r") || 0==strcmp(fm,"rb")){
        MPImode = MPI_MODE_RDONLY;
        /* MESSAGE("Mode = MPI_MODE_RDONLY");fflush(stdout); // DEBUG */
    }else if (  0==strcmp(fm,"w") || 0==strcmp(fm,"wb") ||
                0==strcmp(fm,"a") || 0==strcmp(fm,"ab") ||
                0==strcmp(fm,"a+")|| 0==strcmp(fm,"ab+")||
                0==strcmp(fm,"a+b")){
        MPImode = MPI_MODE_WRONLY | MPI_MODE_CREATE;
        /* MESSAGE("Mode = MPI_MODE_RDONLY | MPI_MODE_CREATE");fflush(stdout); // DEBUG */
    }else if (  0==strcmp(fm,"r+")|| 0==strcmp(fm,"rb+") ||
                0==strcmp(fm,"r+b")){
        MPImode = MPI_MODE_RDWR;
        /* MESSAGE("Mode = MPI_MODE_RDWR");fflush(stdout); // DEBUG */
    }else if (  0==strcmp(fm,"w+")|| 0==strcmp(fm,"wt") ||
                0==strcmp(fm,"w+b")){
        MPImode = MPI_MODE_RDWR | MPI_MODE_CREATE;
        /* MESSAGE("Mode = MPI_MODE_RDWR | MPI_MODE_CREATE");fflush(stdout); // DEBUG */
    }else{
        ERROR("Could not create equivalent MPI file access mode given \"%s\"\n", fm);
    }

    return MPImode;
}
#endif
```

Listing C.29: *sequence.c* — The `getMPIFileMode()` function.

```
#if MPI
int acceptq (const char *name,const char *mode,const char *deflt,int runHere,sequence *q, char *w) {
    deviceCommunicator(runHere, &(q->comm));
    q->mpiMode = getMPIFileMode(mode);
    q->f = NULL; /*  Prevents error on first close. */

    if(runHere){
#else
int acceptq (const char *name,const char *mode,const char *deflt,sequence *q, char *w) {
#endif
    char *p;

    if NOT(accepts(name,&(q->mask[0]),deflt,w)) return 0;

    if (0==strcmp(q->mask,"") || 0==strcmp(q->mask,null)) {
        strcpy(q->name,null);
        q->f=NULL;
        MESSAGE("\x01%s\"%s\"%c",name,NULLFILE,SEPARATORS[0]);
        return 1;
    }

    if ((q->mask)[0]=='*') {q->mute=0;memmove((q->mask),(q->mask)+1,MAXPATH-1);} else {q->mute=1;}

    /*  If mask doesn't contain format specifier, test literal filename. */
    if NOT(strchr(q->mask,'%')) {
        strcpy(q->name,q->mask);
        strcpy(q->mode,mode);
#if MPI
        if (MPI_File_open(q->comm, q->name, q->mpiMode, MPI_INFO_NULL, &(q->f)) != MPI_SUCCESS){
            int devRank;
            MPI_Comm_rank(q->comm,&devRank);
            if(devRank==0){
                ERROR2("could not open \"%s\" in mode \"%s\".
                        Check that the directory exists and that you have the necessary permissions.", q->name,mode);
            }else{
                return 0;
            }
        }
#else
        if NOT(q->f=fopen(q->name,q->mode)) ERROR2("could not open \"%s\" in mode \"%s\".
            Check that the directory exists and that you have the necessary permissions.",q->name,q->mode);
#endif
        return 1;
    }
```

Listing C.30: *sequence.c* — The `acceptq()` function.

```
/*  Point type specific to ppmout (i.e. 3 chars per pixel). */
MPI_Datatype pointType;
mpi_errno = MPI_Type_contiguous(3,MPI_CHAR,&pointType);
CHECK_MPI_SUCCESS("Couldn't create pixel type (pointType).")
mpi_errno = MPI_Type_commit(&pointType);
CHECK_MPI_SUCCESS("Couldn't commit pixel type (pointType).")

/*  Generic 3D filetype based on space. */
/*  Filetype */
MPI_Datatype filetype;
int count;

#define NDIMS 3
int sizes[NDIMS],subsizes[NDIMS],starts[NDIMS];
```

```
/*  Image dimensions. */
int xsize = (dev->s.global_x1 - dev->s.global_x0) + 1;
int ysize = (dev->s.global_y1 - dev->s.global_y0) + 1;
int zsize = (dev->s.global_z1 - dev->s.global_z0) + 1;

/*  Local buffer dimensions */
int local_xsize = (dev->s.x1 - dev->s.x0) + 1;
int local_ysize = (dev->s.y1 - dev->s.y0) + 1;
int local_zsize = (dev->s.z1 - dev->s.z0) + 1;

sizes[0] = zsize;
sizes[1] = ysize;
sizes[2] = xsize;

subsizes[0] = local_zsize;
subsizes[1] = local_ysize;
subsizes[2] = local_xsize;
count = local_xsize*local_ysize*local_zsize;

starts[0] = dev->s.z0 - dev->s.global_z0;
starts[1] = dev->s.y0 - dev->s.global_y0;
starts[2] = dev->s.x0 - dev->s.global_x0;

mpi_errno = MPI_Type_create_subarray(NDIMS, sizes, subsizes, starts, MPI_ORDER_C, pointType, &filetype);
CHECK_MPI_SUCCESS("Couldn't define filetype.")
mpi_errno = MPI_Type_commit(&filetype);
CHECK_MPI_SUCCESS("Couldn't commit filetype.")

S->pointType = pointType;
S->filetype = filetype;
S->count = count;
S->xsize = xsize;
S->ysize = ysize;
S->zsize = zsize;
S->local_xsize = local_xsize;
S->local_ysize = local_ysize;
S->local_zsize = local_zsize;
```

Listing C.31: *ppmout.c* — Defining the MPI derived datatype for the `ppmout` device's file view.

```
RUN_HEAD(ppmout)
    DEVICE_CONST(int,append)
    DEVICE_VAR(sequence,file)
    DEVICE_CONST(int,r) DEVICE_CONST(real,r0) DEVICE_CONST(real,r1)
    DEVICE_CONST(int,g) DEVICE_CONST(real,g0) DEVICE_CONST(real,g1)
    DEVICE_CONST(int,b) DEVICE_CONST(real,b0) DEVICE_CONST(real,b1)
    DEVICE_CONST(int,bgr)
    DEVICE_CONST(int,bgg)
    DEVICE_CONST(int,bgb)

    int x, y, z;
    int nx = (s.x1-s.x0) + 1;
    int ny = (s.y1-s.y0) + 1;
    int nz = (s.z1-s.z0) + 1;
 #if MPI
    DEVICE_CONST(MPI_Datatype,filetype)
    DEVICE_CONST(MPI_Datatype,pointType)
    DEVICE_CONST(int,count)
    DEVICE_CONST(int,xsize)
    DEVICE_CONST(int,ysize)
    DEVICE_CONST(int,zsize)
    DEVICE_CONST(int,local_xsize)
    DEVICE_CONST(int,local_ysize)
    DEVICE_CONST(int,local_zsize)
    char buf[local_zsize][local_ysize][local_xsize][3]; /*  3 is for RGB. */
    char header[100];
    int headerLength;

    /*  For displacement */
    MPI_Aint lb;
    MPI_Aint char_extent;
    MPI_Offset headerSkip;
    MPI_Type_get_extent(MPI_CHAR,&lb, &char_extent);
    MPI_Offset prealloc_size;
#endif
    if (!file->f) return 1;
    DEBUG("Here we are in %s\n",__FILE__);
    if (!append) nextq(file); DEBUG("nextq finished ok\n");
```

Listing C.32: *ppmout.c* — The beginning of the `ppmout` Run function.

```
#if MPI
    sprintf (header,"P6\n%d %d\n%d\n",xsize,ysize,MAXCHAR);
    headerLength = strlen(header);

    /* Preallocate file size */
    prealloc_size = headerLength + xsize*ysize*zsize*3;
    MPI_File_preallocate(file->f,prealloc_size);

    /*  Set view with no displacement (top of file). */
    mpi_errno = MPI_File_set_view(file->f, 0, MPI_CHAR, MPI_CHAR, "native", MPI_INFO_NULL);
    CHECK_MPI_SUCCESS("Couldn't set view for header.")
```

```
    /*  First process in the space writes the header. */
    if(s.x0 == s.global_x0 && s.y0 == s.global_y0 && s.z0 == s.global_z0){
        mpi_errno = MPI_File_write(file->f, header, headerLength, MPI_CHAR, MPI_STATUS_IGNORE);
        CHECK_MPI_SUCCESS("Couldn't write to file.")
    }

    /*  Set view for collective writing. Displace by length of header. */
    headerSkip = char_extent * headerLength;
    mpi_errno = MPI_File_set_view(file->f, headerSkip, pointType, filetype, "native", MPI_INFO_NULL);
    CHECK_MPI_SUCCESS("Couldn't set view for writing.")
#else
    fprintf (file->f,"P6\n%d %d\n%d\n",nx,ny,MAXCHAR);
#endif
```

Listing C.33: *ppmout.c* — Writing the ppm file header.

```
#if MPI
    for(z=0;z<local_zsize;z++) {
        for(y=local_ysize-1;y>=0;y--) {
            for(x=0;x<local_xsize;x++) {
                if(!GEOMETRY_ON || isTissue(s.x0+x,s.y0+y,s.z0+z)){
                    buf[z][y][x][R] = Byte(s.x0+x,s.y0+y,s.z0+z,r,r0,r1);
                    buf[z][y][x][G] = Byte(s.x0+x,s.y0+y,s.z0+z,g,g0,g1);
                    buf[z][y][x][B] = Byte(s.x0+x,s.y0+y,s.z0+z,b,b0,b1);
                }else{ /*  Void. Use background colour. */
                    buf[z][y][x][R] = (unsigned) bgr;
                    buf[z][y][x][G] = (unsigned) bgg;
                    buf[z][y][x][B] = (unsigned) bgb;
                } /*  else */
            } /*  for x */
        } /*  for y */
    } /*  for z */

    mpi_errno = MPI_File_write(file->f, buf, count, pointType, MPI_STATUS_IGNORE);
    CHECK_MPI_SUCCESS("Couldn't write to file.")
#else
    /*  Sequential version */
    for(z=0;z<nz;z++){
        for(y=0;y<ny;y++) {
            for(x=0;x<nx;x++) {
                if(!GEOMETRY_ON || isTissue(s.x0+x,s.y0+y,s.z0+z)){
                    putc(Byte(s.x0+x,s.y0+y,s.z0+z,r,r0,r1),file->f);
                    putc(Byte(s.x0+x,s.y0+y,s.z0+z,g,g0,g1),file->f);
                    putc(Byte(s.x0+x,s.y0+y,s.z0+z,b,b0,b1),file->f);
                }else{ /*  Void. Use background colour. */
                    putc((unsigned)bgr,file->f);
                    putc((unsigned)bgg,file->f);
                    putc((unsigned)bgb,file->f);
                }
            } /*  for x */
        } /*  for y */
        if (append) { fflush(file->f); DEBUG("file fflushed\n"); }
    } /*  for z */
#endif
RUN_TAIL(ppmout)
```

Listing C.34: *ppmout.c* — Writing the image data to file.

APPENDIX

D

MPI PERFORMANCE SCRIPTS

```
// Computation parameters
<fhn.par>

state xmax=302 ymax=302 zmax=302 vmax=3;

def real begin;
def real end;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,100);
};

// Stimulus
k_func when=begin x1=200 pgm={u[u] = 1.7;};
k_func when=begin x0=201 pgm={u[u] = -1.7;};
k_func when=begin y1=200 pgm={u[v] = 0.7;};
k_func when=begin y0=201 pgm={u[v] = -0.7;};

// The computation
diff v0=[u] v1=[i] D=D hx=hx;
euler v0=[u] v1=[v] ht=ht ode=fhncubpar rest=10000 par={epsu=eps epsv=eps bet=bet gam=gam Iu=

ppmout
    file="out/[0]_%04d.ppm" mode="w"
    r=[u] r0=umin r1=umax
    g=[u] g0=umin g1=umax
    b=[v] b0=vmin b1=vmax
;

record
    x0=35 x1=35
    y0=35 y1=35
    z0=35 z1=35
    v0=[u] v1=[v]
    filehead="SCALING TEST"
```

```
    file=out/[0].rec mode=append
;

dump append=0 file=out/[0].dmp;

stop when=end;
stop when=end;

end;
```

Listing D.1: *bigbox_fhn.bbs* — Script used to test MPI scaling on HECToR. $300 \times 300 \times 300$ mesh, using FitzHugh-Nagumo kinetics.

```
<lrd.par>

def int neqn [iext];

state xmax=302 ymax=302 zmax=302 vmax=[iext]+1;

def real begin;
def real end;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,100);
};

// Stimulus
k_func when=begin x0=18 x1=22 y0=18 y1=22 z0=18 z1=22 pgm={u[V]=Vmax;};

// The computation
diff v0=[V] v1=[iext] D=D hx=hx;
euler v0=[V] v1=neqn-1 ht=ht ode=lrd rest=10000 par={ht=ht Iu=@[iext]};

ppmout
    file="out/[0]_%04d.ppm" mode="w"
    r=[V]       r0=Vmin     r1=Vmax
    g=[xi]      g0=0        g1=1
    b=[dvdt]    b0=0        b1=1
;

record
    x0=35 x1=35
    y0=35 y1=35
    z0=35 z1=35
    v0=[V] v1=neqn-1
    filehead="SCALING TEST"
    file=out/[0].rec mode=append
;

dump append=0 file=out/[0].dmp;

stop when=end;
stop when=end;
```

```
end;
```

Listing D.2: *bigbox_lrd.bbs* — Script used to test MPI scaling on HECToR. $300 \times 300 \times 300$ mesh, using Luo-Rudy kinetics.

APPENDIX

E

CODE LISTINGS FOR CHAPTER 4

```
#define geom_ind(x,y,z,v)   (                       \
    ((x + ONE) - local_xmin)*geom_vmax_zmax_ymax +  \
    ((y + TWO) - local_ymin)*geom_vmax_zmax+        \
    ((z + TRI) - local_zmin)*geom_vmax+(v)          \
)
```

Listing E.1: *state.h* — MPI version of `geom_ind`, the geometry data indexing function.

```
/* Tests the geometry status (TISSUE/VOID) of a point.
 * Takes (x,y,z) coordinates and evaluates to true or false.
 */
#define isTissue(x,y,z) (                                                       \
    (   x>=SPACE_DEFAULT_X0 && x<=SPACE_DEFAULT_X1  &&                          \
        y>=SPACE_DEFAULT_Y0 && y<=SPACE_DEFAULT_Y1  &&                          \
        z>=SPACE_DEFAULT_Z0 && z<=SPACE_DEFAULT_Z1) &&                          \
        (!GEOMETRY_ON || Geom[geom_ind(x,y,z,GEOM_STATUS)] == (real)GEOM_TISSUE)    \
    )
```

Listing E.2: *state.h* — The `isTissue` function macro.

```
DEVICE_CONST(real,D)
real gam=D/(hx*hx);
for(x=s.x0;x<=s.x1;x++) {
    for(y=s.y0;y<=s.y1;y++){
        for(z=s.z0;z<=s.z1;z++){
            if(isTissue(x,y,z)){
                sum = 0;
                    u=New+ind(x,y,z,s.v0);
                value = u[0];
                if(dim>=1){
                    sum += isTissue(x-1,y  ,z  )? u[-DX] : value;
                    sum += isTissue(x+1,y  ,z  )? u[ DX] : value;
                }
                if(dim>=2){
                    sum += isTissue(x  ,y-1,z  )? u[-DY] : value;
                    sum += isTissue(x  ,y+1,z  )? u[ DY] : value;
                }
                if(dim>=3){
                    sum += isTissue(x  ,y  ,z-1)? u[-DZ] : value;
                    sum += isTissue(x  ,y  ,z+1)? u[ DZ] : value;
                }
                u[DV*(s.v1-s.v0)] = gam * ( sum - (2*dim*u[0]) );
            } /* if isTissue() */
        } /* for z */
    } /* for y */
} /* for x */
```

Listing E.3: *diff.c* — Laplacian with implicit Neumann boundary conditions. The function automatically accommodates for one-, two- and three-dimensional meshes.

```
/* Fibres of central point */
f[1] =  Geom[geom_ind(x,y,z,GEOM_FIBRE_1)];
f[2] =  Geom[geom_ind(x,y,z,GEOM_FIBRE_2)];
f[3] =  Geom[geom_ind(x,y,z,GEOM_FIBRE_3)];

for(i=1;i<=3;i++){
    for(j=1;j<=3;j++){
        /* \mathbf{D} =
         * D_\perp \mathbf{I} + (D_\parallel - D\perp)
```

```
         * \mathbf{A}\mathbf{A}^T
         */
        S->points[ind].D[i][j] = ((Dpar-Dtrans)*f[i]*f[j]) + ((i==j)?Dtrans:0);
    }
}
```

Listing E.4: *diff.c* — Precomputing **D**.

```
for(i=1;i<=3;i++){
    if(
        !isTissue( x+(i==1?1:0), y+(i==2?1:0), z+(i==3?1:0) ) ||
        !isTissue( x-(i==1?1:0), y-(i==2?1:0), z-(i==3?1:0) )
    ){
        for(j=1;j<=3;j++){dD[i][j]=0.0;}
    }else{
        for(j=1;j<=3;j++){
            /* d^{11}_{+..} - d^{11}_{-..}
             * = (D_\parallel - D_\perp)(a_i a_j)_{+..} -(a_i a_j)_{-..}
             */
            nx = x-(i==1?1:0);
            ny = y-(i==2?1:0);
            nz = z-(i==3?1:0);
            px = x+(i==1?1:0);
            py = y+(i==2?1:0);
            pz = z+(i==3?1:0);
            dD[i][j] =
                (Dpar - Dtrans) *
                (
                    (Geom[geom_ind(px,py,pz,(GEOM_FIBRE_1+j-1) )] *
                     Geom[geom_ind(px,py,pz,(GEOM_FIBRE_1+j-1) )] )-
                    (Geom[geom_ind(nx,ny,nz,(GEOM_FIBRE_1+j-1) )] *
                     Geom[geom_ind(nx,ny,nz,(GEOM_FIBRE_1+j-1) )] )
                );
        } /* for j */
    } /* else */
} /* for i */

for(i=1;i<=3;i++){
    S->points[ind].c[i] = 0;
    for(j=1;j<=3;j++){
        S->points[ind].c[i] += dD[j][i];
    }
}
```

Listing E.5: *diff.c* — Precomputing differences of the diffusion tensor.

```
DEVICE_ARRAY(AnisoPoint, points)
DEVICE_CONST(int, numTissuePoints)
int ind;
real sum;

for(ind=0;ind<numTissuePoints;ind++){
    x = points[ind].x;
    y = points[ind].y;
    z = points[ind].z;

    /* Centre point */
    u=New+ind(x,y,z,s.v0);
    sum = 0;

    u=New+ind(x,y,z,s.v0);
    if( isTissue(x+1,y  ,z  ))                              sum += 4 *  points[ind].D[1][1] * (u[+DX        ] - u[     0     ]);
    if( isTissue(x-1,y  ,z  ))                              sum += 4 *  points[ind].D[1][1] * (u[-DX        ] - u[     0     ]);
    if( isTissue(x  ,y+1,z  ))                              sum += 4 *  points[ind].D[2][2] * (u[    +DY    ] - u[     0     ]);
    if( isTissue(x  ,y-1,z  ))                              sum += 4 *  points[ind].D[2][2] * (u[    -DY    ] - u[     0     ]);
    if( isTissue(x  ,y  ,z+1))                              sum += 4 *  points[ind].D[3][3] * (u[          DZ] - u[     0     ]);
    if( isTissue(x  ,y  ,z-1))                              sum += 4 *  points[ind].D[3][3] * (u[        -DZ] - u[     0     ]);

    if(isTissue(x+1,y  ,z  ) && isTissue(x-1,y  ,z  ))      sum +=      points[ind].c[1]    * (u[+DX        ] - u[-DX        ]);
    if(isTissue(x  ,y+1,z  ) && isTissue(x  ,y-1,z  ))      sum +=      points[ind].c[2]    * (u[    +DY    ] - u[    -DY    ]);
    if(isTissue(x  ,y  ,z+1) && isTissue(x  ,y  ,z-1))      sum +=      points[ind].c[3]    * (u[        +DZ] - u[        -DZ]);

    if(isTissue(x+1,y+1,z  ) && isTissue(x+1,y-1,z  ))      sum += 2 *  points[ind].D[1][2] * (u[+DX +DY    ] - u[+DX -DY    ]);
    if(isTissue(x-1,y-1,z  ) && isTissue(x-1,y+1,z  ))      sum += 2 *  points[ind].D[1][2] * (u[-DX -DY    ] - u[-DX +DY    ]);
    if(isTissue(x+1,y  ,z+1) && isTissue(x+1,y  ,z-1))      sum += 2 *  points[ind].D[1][3] * (u[+DX     +DZ] - u[+DX     -DZ]);
    if(isTissue(x-1,y  ,z-1) && isTissue(x-1,y  ,z+1))      sum += 2 *  points[ind].D[1][3] * (u[-DX     -DZ] - u[-DX     +DZ]);
    if(isTissue(x  ,y+1,z+1) && isTissue(x  ,y+1,z-1))      sum += 2 *  points[ind].D[2][3] * (u[    +DY +DZ] - u[    +DY -DZ]);
    if(isTissue(x  ,y-1,z-1) && isTissue(x  ,y-1,z+1))      sum += 2 *  points[ind].D[2][3] * (u[    -DY -DZ] - u[    -DY +DZ]);

    u[DV*(s.v1-s.v0)] = sum / (4*hx*hx);

} /* for ind */
```

Listing E.6: *diff.c* — Computing the Laplacian with anisotropy.

APPENDIX

F

GEOMETRY VERIFICATION & PERFORMANCE SCRIPTS

```
// Computation parameters
<fhn.par>

state   geometry=ffr_yslice.bbg
    vmax=3
    normaliseVectors=1
    correctBoundaries=1
;

def real begin;
def real end;
def real out;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,100000);
    out   = eq(mod(t,10),0);
};

// Cross-field stimulus
k_func when=begin x1=25 pgm={u[u] = 1.7;};
k_func when=begin x0=26 pgm={u[u] = -1.7;};
k_func when=begin y1=25 pgm={u[v] = 0.7;};
k_func when=begin y0=26 pgm={u[v] = -0.7;};

// The computation
diff
    v0=[u] v1=[i]
    D=D
    hx=hx
;
euler
    v0=[u] v1=[v]
    ht=ht ode=fhncubpar rest=10000
    par={epsu=eps epsv=eps bet=bet gam=gam Iu=@[i]}
;
```

```
ppmout
    when=out
    file="ppm/%04d.ppm" mode="w"
    r=[u] r0=umin r1=umax bgr=255
    g=[v] g0=vmin g1=vmax bgg=255
    b=[i] b0=0 b1=255 bgb=255
;

stop when=end;
end;
```

Listing F.1: Beatbox script describing cross-fields stimulation in a two-dimensional slice of rabbit ventricles, with isotropic diffusion.

```
// Computation parameters
<fhn.par>

state   geometry=ffr_yslice.bbg
    vmax=3
    normaliseVectors=1
    correctBoundaries=1
    anisotropy=1
;

def real begin;
def real end;
def real out;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,100000);
    out   = eq(mod(t,10),0);
};

// Cross-field stimulus
k_func when=begin x1=25 pgm={u[u] = 1.7;};
k_func when=begin x0=26 pgm={u[u] = -1.7;};
k_func when=begin y1=25 pgm={u[v] = 0.7;};
k_func when=begin y0=26 pgm={u[v] = -0.7;};

// The computation
diff
    v0=[u] v1=[i]
    Dpar=D Dtrans=D/4
    hx=hx
;
euler
    v0=[u] v1=[v]
    ht=ht ode=fhncubpar rest=10000
    par={epsu=eps epsv=eps bet=bet gam=gam Iu=@[i]}
;

ppmout
```

```
    when=out
    file="ppm/%04d.ppm" mode="w"
    r=[u] r0=umin r1=umax bgr=255
    g=[v] g0=vmin g1=vmax bgg=255
    b=[i] b0=0 b1=255 bgb=255
;

stop when=end;
end;
```

Listing F.2: Beatbox script describing cross-fields stimulation in a two-dimensional slice of rabbit ventricles, with anisotropic diffusion.

```
// Computation parameters
<fhn.par>

state
    geometry=ffr.bbg
    vmax=3
    normaliseVectors=1
    correctBoundaries=1
;

def real begin;
def real end;
def real out;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,100000);
    out   = eq(mod(t,10),0);
};

// Stimulus
k_func when=begin x1=25 pgm={u[u] = 1.7;};
k_func when=begin x0=26 pgm={u[u] = -1.7;};
k_func when=begin y1=25 pgm={u[v] = 0.7;};
k_func when=begin y0=26 pgm={u[v] = -0.7;};

// The computation
diff v0=[u] v1=[i] D=D hx=hx;
euler
    v0=[u] v1=[v]
    ht=ht ode=fhncubpar rest=10000
    par={epsu=eps epsv=eps bet=bet gam=gam Iu=@[i]}
;

ppmout
    when=out
    file="ppm/%04d.ppm" mode="w"
    // Blue for void points.
    r=[u] r0=umin r1=umax bgr=0
    g=[v] g0=vmin g1=vmax bgg=0
    b=[i] b0=0 b1=255 bgb=255
```

```
;

stop when=end;
stop when=end;

end;
```

Listing F.3: Beatbox script describing cross-field stimulation in rabbit ventricles with isotropic diffusion.

```
// Computation parameters
<fhn.par>
def str p 3;
def str d 4;

state   geometry=ffr.bbg
        vmax=3
        normaliseVectors=1
        correctBoundaries=1
        anisotropy=1
;

def real begin;
def real end;
def real out;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,10000);
    out   = eq(mod(t,100),0);
};

// Stimulus
k_func when=begin x1=25 pgm={u[u] = 1.7;};
k_func when=begin x0=26 pgm={u[u] = -1.7;};
k_func when=begin y1=25 pgm={u[v] = 0.7;};
k_func when=begin y0=26 pgm={u[v] = -0.7;};

// The computation
diff v0=[u] v1=[i] Dpar=D Dtrans=D/4 hx=hx;
euler
    v0=[u] v1=[v]
    ht=ht
    ode=fhncubpar
    rest=10000
    par={epsu=eps epsv=eps bet=bet gam=gam Iu=@[i]}
;

record file=[0].rec when=end v0=0 v1=1;
ppmout
    when=out
    file="ppm/%04d.ppm"
    mode="w"
    r=[u] r0=umin r1=umax
```

```
    g=[v] g0=vmin g1=vmax
    b=[i] b0=0 b1=255
;

stop when=end;
stop when=end;

end;
```

Listing F.4: Beatbox script describing cross-field stimulation in rabbit ventricles with anisotropic diffusion.

```
// Computation parameters
<fhn.par>

// Roughly same number of tissue points as ffr.bbg
state xmax=69 ymax=80 zmax=92 vmax=3;

def real begin;
def real end;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,1000);
};

// Stimulus
k_func when=begin x1=35 pgm={u[u] = 1.7;};
k_func when=begin x0=36 pgm={u[u] = -1.7;};
k_func when=begin y1=35 pgm={u[v] = 0.7;};
k_func when=begin y0=36 pgm={u[v] = -0.7;};

// The computation
diff v0=[u] v1=[i] D=D hx=hx;
euler
    v0=[u] v1=[v]
    ht=ht
    ode=fhncubpar
    rest=10000
    par={epsu=eps epsv=eps bet=bet gam=gam Iu=@[i]}
;

ppmout
    file="out/[0]_%04d.ppm" mode="w"
    r=[u] r0=umin r1=umax
    g=[v] g0=vmin g1=vmax
    b=[u] b0=200  b1=255
;

record
    x0=35 x1=35
    y0=35 y1=35
    z0=35 z1=35
    v0=[u] v1=[v]
```

```
    filehead="SCALING TEST"
    file=out/[0].rec mode=append
;

dump append=0 file=out/[0].dmp;

stop when=end;
stop when=end;

end;
```

Listing F.5: Scaling script 1 — Cuboid mesh containing approximately the same number of tissue points as the rabbit ventricle mesh ($69 \times 80 \times 92$). Isotropic diffusion. FitzHugh-Nagumo kinetics

```
// Computation parameters
<fhn.par>

state
    geometry=ffr.bbg
    vmax=3
    normaliseVectors=1
    anisotropy=0
;

def real begin;
def real end;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,1000);
};

// Stimulus
k_func when=begin x1=75 pgm={u[u] = 1.7;};
k_func when=begin x0=76 pgm={u[u] = -1.7;};
k_func when=begin y1=75 pgm={u[v] = 0.7;};
k_func when=begin y0=76 pgm={u[v] = -0.7;};

// The computation
diff v0=[u] v1=[i] D=D hx=hx;
euler
    v0=[u] v1=[v]
    ht=ht
    ode=fhncubpar
    rest=10000
    par={epsu=eps epsv=eps bet=bet gam=gam Iu=@[i]}
;

ppmout
    file="out/[0]_%04d.ppm" mode="w"
    r=[u] r0=umin r1=umax bgr=0
    g=[v] g0=vmin g1=vmax bgg=0
    b=[u] b0=200  b1=201  bgb=255
```

```
;

record
    x0=35 x1=35
    y0=35 y1=35
    z0=35 z1=35
    v0=[u] v1=[v]
    filehead="SCALING TEST"
    file=out/[0].rec mode=append
;

dump append=0 file=out/[0].dmp;

stop when=end;
stop when=end;

end;
```

Listing F.6: Scaling script 2 — Rabbit ventricle mesh ($119 \times 109 \times 131$). Isotropic diffusion. FitzHugh-Nagumo kinetics

```
// Computation parameters
<fhn.par>

state
    geometry=ffr.bbg
    vmax=3
    normaliseVectors=1
    anisotropy=1
;

def real begin;
def real end;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,1000);
};

// Stimulus
k_func when=begin x1=75 pgm={u[u] = 1.7;};
k_func when=begin x0=76 pgm={u[u] = -1.7;};
k_func when=begin y1=75 pgm={u[v] = 0.7;};
k_func when=begin y0=76 pgm={u[v] = -0.7;};

// The computation
diff v0=[u] v1=[i] Dpar=D Dtrans=D/4 hx=hx;
euler
    v0=[u] v1=[v]
    ht=ht
    ode=fhncubpar
    rest=10000
    par={epsu=eps epsv=eps bet=bet gam=gam Iu=@[i]}
;
```

```
ppmout
    file="out/[0]_%04d.ppm" mode="w"
    r=[u] r0=umin r1=umax bgr=0
    g=[v] g0=vmin g1=vmax bgg=0
    b=[u] b0=200  b1=201  bgb=255
;

record
    x0=35 x1=35
    y0=35 y1=35
    z0=35 z1=35
    v0=[u] v1=[v]
    filehead="SCALING TEST"
    file=out/[0].rec mode=append
;

dump append=0 file=out/[0].dmp;

stop when=end;
stop when=end;

end;
```

Listing F.7: Scaling script 2 — Rabbit ventricle mesh ($119 \times 109 \times 131$). Anisotropic diffusion. FitzHugh-Nagumo kinetics

```
// Computation parameters
<lrd.par>

def int neqn [iext];

// Roughly same number of tissue points as ffr.bbg
state xmax=69 ymax=80 zmax=92 vmax=[iext]+1;

def real begin;
def real end;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,1000);
};

// Stimulus
k_func when=begin x0=18 x1=22 y0=18 y1=22 z0=18 z1=22 pgm={u[V]=Vmax;};


// The computation
diff v0=[V] v1=[iext] D=D hx=hx;
euler v0=[V] v1=neqn-1 ht=ht ode=lrd rest=10000 par={ht=ht Iu=@[iext]};

ppmout
    file="out/[0]_%04d.ppm" mode="w"
    r=[V] r0=Vmin r1=Vmax
```

```
    g=[xi]      g0=0        g1=1
    b=[dvdt]    b0=0        b1=1
;

record
    x0=35 x1=35
    y0=35 y1=35
    z0=35 z1=35
    v0=[V] v1=neqn-1
    filehead="SCALING TEST"
    file=out/[0].rec mode=append
;

dump append=0 file=out/[0].dmp;

stop when=end;
stop when=end;

end;
```

Listing F.8: Scaling script 1 — Cuboid mesh containing approximately the same number of tissue points as the rabbit ventricle mesh ($69 \times 80 \times 92$). Isotropic diffusion. Luo-Rudy kinetics

```
// Computation parameters
<lrd.par>

def int neqn [iext];

state
    geometry=ffr.bbg
    vmax=[iext]+1
    normaliseVectors=1
    anisotropy=0
;

def real begin;
def real end;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,1000);
};

// Stimulus
k_func when=begin x0=18 x1=22 y0=18 y1=22 z0=18 z1=22 pgm={u[V]=Vmax;};

// The computation
diff v0=[V] v1=[iext] D=D hx=hx;
euler v0=[V] v1=neqn-1 ht=ht ode=lrd rest=10000 par={ht=ht Iu=@[iext]};

ppmout
    file="out/[0]_%04d.ppm" mode="w"
    r=[V]       r0=Vmin     r1=Vmax
    g=[xi]      g0=0        g1=1
```

```
    b=[dvdt]    b0=0        b1=1
;

record
    x0=35 x1=35
    y0=35 y1=35
    z0=35 z1=35
    v0=[V] v1=neqn-1
    filehead="SCALING TEST"
    file=out/[0].rec mode=append
;

dump append=0 file=out/[0].dmp;

stop when=end;
stop when=end;

end;
```

Listing F.9: Scaling script 2 — Rabbit ventricle mesh ($119 \times 109 \times 131$). Isotropic diffusion. Luo-Rudy kinetics

```
// Computation parameters
<lrd.par>

def int neqn [iext];

state
    geometry=ffr.bbg
    vmax=[iext]+1
    normaliseVectors=1
    anisotropy=1
;

def real begin;
def real end;

k_func name=timing nowhere=1 pgm={
    begin = eq(t,0);
    end   = ge(t,1000);
};

// Stimulus
k_func when=begin x0=18 x1=22 y0=18 y1=22 z0=18 z1=22 pgm={u[V]=Vmax;};

// The computation
diff v0=[V] v1=[iext] Dpar=D Dtrans=D/4 hx=hx;
euler v0=[V] v1=neqn-1 ht=ht ode=lrd rest=10000 par={ht=ht Iu=@[iext]};

ppmout
    file="out/[0]_%04d.ppm" mode="w"
    r=[V]       r0=Vmin     r1=Vmax
    g=[xi]      g0=0        g1=1
    b=[dvdt]    b0=0        b1=1
```

```
;

record
    x0=35 x1=35
    y0=35 y1=35
    z0=35 z1=35
    v0=[V] v1=neqn-1
    filehead="SCALING TEST"
    file=out/[0].rec mode=append
;

dump append=0 file=out/[0].dmp;

stop when=end;
stop when=end;

end;
```

Listing F.10: Scaling script 2 — Rabbit ventricle mesh ($119 \times 109 \times 131$). Anisotropic diffusion. Luo-Rudy kinetics

APPENDIX

G

BEATBOX SCRIPTING GUIDE

A Beatbox simulation is defined by the user in the form of a script. A script is a plain text file that specifies the parameters of the simulation. Any script can be run on any of Beatbox's supported hardware platforms. This section introduces Beatbox scripts by first examining the scripting language, before discussing common applications. Unlike interpreted 'scripting' languages such as PHP or Python, Beatbox scripts are not run during the simulation. The script is read once to build the simulation, after which script code does not define the flow of execution.

G.1 The Beatbox Scripting Language

Similar to other programming languages, Beatbox scripts allow the user to define variables and macros, include external code and make calls to the operating system. The Beatbox scripting language also provides a small library of arithmetic and logical functions. The user may also improve the legibility of their code, or disable portions of the code using comments. Each of these features is discussed below.

A Beatbox script comprises a series of *commands*. A command takes the form of any number of lines of script code beginning with a recognised keyword and ending with a semicolon (`;`). The following code excerpt shows examples of three commands:

```
def int sideLength 26;

state xmax=sideLength, ymax=sideLength, zmax=sideLength, vmax=3;

euler
    v1=[iext]
    ode=lrd
    par={ht=ht IV=@24}
;
```

The types of command in a Beatbox script are listed below. Each command describes an action to be taken by Beatbox, with the keyword at the beginning of the command being the verb.

- `rem` — A 'remark'. Marks code as a comment. These are ignored by the interpreter.
- `if` — Allows a script command to be conditionally read. This does not affect the flow of execution as the simulation is run, but operates much like conditional compilation.

- `def` — Defines a variable or macro.
- `state` — Sets the dimensions of the simulation medium.
- `screen` — Sets parameter for on-screen display.
- Device calls — Adds a device to the ring of devices. The keyword in a device call may be any of the device names listed in *devlist.h*, e.g. `euler`.
- `end` — Marks the end of the script.

G.2 Preprocessing

As each line of the script is parsed, Beatbox preprocesses the code in a manner similar to the C Preprocessor [see Kernighan and Ritchie, 1988, chap. 4]. The Beatbox preprocessor handles four tasks: skipping comments, including other Beatbox scripts, calling system commands and expanding string macros. Each of these are discussed below.

G.2.1 Skipping Comments

Comments can be added to Beatbox scripts in three ways:

```
/*
    Multi-line
    C style
    comment
*/

// Single-line C++ style comment

rem 'Remark' command-style comment, which must end with a semicolon;
```

Code in a comment is ignored by the parser.

G.2.2 Including Other Beatbox Scripts

Where the relative path to a Beatbox script file is enclosed in in angle brackets (`< >`), the content of the referenced file will be read and inserted at that location. Unlike in the C programming language, the name of the file in angle brackets is not preceded by an `#include` command. This can be used to maintain consistency across a number of simulations, or to reduce code redundancy. Beatbox replaces a filename in angle brackets with the file's entire contents. For example, given a script called *useful.qui*:

```
// Here is a lot of useful code...
```

Listing G.1: *useful.qui*

and a script that includes *useful.qui* as follows:

```
// This is my own script
<useful.qui>
// Here's some more of my own code.
```

Listing G.2: A script that includes *useful.qui*.

The result after preprocessing is shown below:

```
// This is my own script
// Here is a lot of useful code...
// Here's some more of my own code.
```

Listing G.3: Effective resultant script, after *useful.qui* has been included.

G.2.3 Calling System Commands

Beatbox replaces code in backticks (‘‘) with the result of that code when run as a system command, via `system()` (*stdlib.h*). For example ‘`date '+%Y%m%d-%H:%M:%S'`‘ will be replaced with the result of the UNIX `date` command.

G.2.4 Expanding String Macros

String macros allow the user to define reusable strings that can be `pasted` throughout a script. String macros are distinct from variables in that they are expanded once, prior to the script being interpreted and cannot therefore be assigned values other than when they are defined. A string macro is defined as follows:

```
def str <macro name> <value>;
```

where `<macro name>` is a string using letters, numbers, underscores (_) or hyphens (-) and `<value>` is any string not including a semicolon.

When defined, the macro's name, wrapped in brackets (`[ ]`) is associated with its value. When Beatbox finds a macro's name in square brackets, it replaces them with the value. In the following excerpt, the variable `hat` is assigned the value `porkpie`. The variable `headware` is assigned the value `hat`.

```
def str snack porkpie; // Assigned string 'porkpie'.
def str hat [snack]; // Assigned string 'porkpie'.
def str headware hat; // Assigned string 'hat', not value of hat macro.
```

A macro can expand to anything that could be typed in the script. For example, a string macro can be used in place of an `int`, `long` or `real`:

```
def str ninetynine 99; // Assigned string value '99'.
def int number [ninetynine]; // Assigned integer value 99.
```

A number variable cannot, however, be used to define a macro:

```
def int number 99; // Assigned integer value 99.
def str ninetynine number; // Assigned string value 'number'.
```

It is possible to assign several lines of code (excluding semicolons) to a string macro, so this:

```
def str instruction ppmout
when=out file="ppm/%04d.ppm" mode="w"
r=[u] r0=umin r1=umax
g=[v] g0=vmin g1=vmax
b=[i] b0=0 b1=255;
[instruction];
```

is equivalent to:

```
ppmout when=out file="ppm/%04d.ppm" mode="w"
r=[u] r0=umin r1=umax
g=[v] g0=vmin g1=vmax
b=[i] b0=0 b1=255;
```

G.3 Defining Arithmetic Variables with the `def` Command

A variable is defined using the `def` command, like this:

```
def <type> <variable name> <initial value>;
```

where `<type>` is one of Beatbox's pre-defined datatypes:

`int` Integer. Equivalent to `int` in C.

`long` Long integer. Equivalent to `long` in C.

`real` Real number. Equivalent to `double` in C.

`<variable name>` is a string using letters, numbers, underscores (`_`) or hyphens (`-`). Variables defined in this way are accessible in the script and in user-defined functions run by the `k_func` device. `<initial value>` (optional) is an expression that may be evaluated to the correct type. Expressions are discussed in greater detail below.

Variables defined in the script are distinct from the C language variables used in Beatbox's implementation. For clarity, variables defined in the script, using `def` may be referred to as k_variables. Beatbox stores the names of macros wrapped in their brackets (`[]`), therefore it is possible for a script to define macros and variables with the same name.

G.3.1 Expressions

Any numerical value in a Beatbox script, be it a variable definition or parameter, may be specified as a mathematical expression. An expression may be a literal value, such as `5.0`, or the result of some computation, such as `6*5` or `count/2`. Beatbox provides the standard infix mathematical operators: assignment (`=`), addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`). For slightly more complex operations Beatbox provides a collection of functions, the parameter and return types for which are all `real`:

`atan2(a, b)` Returns the arctangent of `b/a`, using the signs of the arguments to compute the quadrant of the return value.

`hypot(a, b)` Returns the square root of the sum of the squares of `a` and `b`.

`tanh(a)` Returns the hyperbolic tangent of `a`.

`erf(a)` Returns the Gauss error function of `a`.

`if(test, then, else)` Returns `then` if `test` evaluates to true, `else` otherwise.

`ifne0(test, then, else)` Returns `then` if `test` is not equal to `0`, `else` otherwise.

`ifeq0(test, then, else)` Returns `then` if `test` is equal to `0`, `else` otherwise.

`ifgt0(test, then, else)` Returns `then` if `test` is strictly greater than `0`, `else` otherwise.

`ifge0(test, then, else)` Returns `then` if `test` is greater than or equal to `0`, `else` otherwise.

`iflt0(test, then, else)` Returns `then` if `test` is strictly less than `0`, `else` otherwise.

`ifle0(test, then, else)` Returns `then` if `test` is less than or equal to `0`, `else` otherwise.

`ifsign(test, neg, zero, pos)` Returns `neg` if `test` is strictly less than `0`, `zero` if `test` is equal to `0`, `pos` if `test` is strictly greater than `0`.

`gt(a, b)` Returns `1.0` if `a` is strictly greater than `b`.

`ge(a, b)` Returns `1.0` if `a` is greater than or equal to `b`.

`lt(a, b)` Returns `1.0` if `a` is strictly less than `b`.

`le(a, b)` Returns `1.0` if `a` is less than or equal to `b`.

`eq(a, b)` Returns `1.0` if `a` and `b` are equal.

`ne(a, b)` Returns `1.0` if `a` and `b` are not equal.

`mod(a, b)` Returns the remainder of the integer division of `a` by `b`.

`max(a, b)` Returns the greater of `a` or `b`.

`min(a, b)` Returns the lesser of `a` or `b`.

`crop(test, min, max)` Returns `min` if `test` is strictly less than `min`. Returns `max` if `test` is strictly less than `max`. Returns `test` otherwise.

`rnd(a, b)` Returns a random real number greater than or equal to `a`, strictly less than `b`.

`u(x,y,z,v)` Returns the value of dynamic variable `v` at point `(x,y,z)` in the mesh. If non-integer values are given for `x` `y` or `z`, the value is linearly interpolated from surrounding points. Non-integer values for `v` are rounded down.

`dudx(x,y,z,v)` Returns the difference in `v` values at the location with x ±`0.5`.

`dudy(x,y,z,v)` Returns the difference in `v` values at the location with y ±`0.5`.

`dudz(x,y,z,v)` Returns the difference in `v` values at the location with z ±`0.5`.

G.4 Defining the Simulation Medium with the `state` Command

The first functional task of the script is to define the mesh. The mesh is stored as a four-dimensional array, corresponding to a three-dimensional, regular Cartesian mesh with an array of dynamic variables at each point. Dynamic variables hold space-dependent values, local to each point in the mesh. The majority of dynamic variables are commonly employed to hold variables of the cell model. The dimensions of the mesh and number of dynamic variables (i.e. the size of the four-dimensional array) are set using the `state` command. `state` has four compulsory parameters; `xmax`, `ymax`, `zmax` and `vmax`. The mesh defined in the example below has 300 points along the x axis and 400 points along the y axis. Since `zmax` is equal to `1`, there is one point along the z axis, making the medium flat on the x-y plane. `vmax` is the number of dynamic variables stored at every point in the mesh.

```
state xmax=300 ymax=400 zmax=1 vmax=3;
```

Listing G.4: Defining a two-dimensional mesh with the `state` command.

Beatbox assumes that the simulation medium will follow a hierarchy of axes; x before y before z, i.e. one-dimensional simulations must use the x axis and two-dimensional simulations must use the x and y axes.

G.4.1 Anatomically Realistic Tissue Geometry

If you would like to use an anatomically realistic mesh for your simulation, you can specify it in the `geometry` parameter of the `state` module. For example, `geometry=ffr.bbg` will select the *ffr.bbg* geometry file. When using a geometry file, there is no need to specify the dimensions of the mesh, in fact Beatbox will complain if you do. You do still need to define `vmax`, however, to suit your chosen RHS module and any other dynamic variables your simulation needs.

Beatbox will also complain if the fibre directions specified in your geometry file are not unit vectors. If you like, Beatbox will normalise the fibre directions for you, if you set the `normaliseVectors` parameter to `1`.

If you would like to model anisotropy, set the `ansiotropy` parameter to `1`. This only works when a geometric mesh is specified. When the anisotropy feature is on, the behaviour and required parameters of some devices may change. In particular, the `diff` device will require two diffusion coefficients, `Dpar` and `Dtrans` instead of the isotropic `D`.

G.5 Device Calls

Following the `state` declaration, a script can add to the *ring of devices* by invoking or 'calling' Beatbox devices. A device call takes the form of the device name, followed by parameters as key-value pairs, separated by spaces. The excerpt below shows the `euler` device being called with some parameters.

```
euler v1=[iext] ht=ht ode=lrd pars={IV=@24} name=Geoff;
```

Each device call will add an instance of the device to the ring of devices. It is possible to instantiate the same device, e.g. `euler`, multiple times and the parameters of each instance will remain independent.

G.5.1 Assigning Device Parameters

Device parameters are assigned following the device name as key-value pairs with the syntax `key=value`. Since spaces separate device parameters, an `str` parameter value that includes spaces, should be enclosed in quotes (`" "`). Expressions assigned to `int`, `long` and `real` parameters will be reduced to literal values when read. An example is shown in Listing G.5.

Some devices may request parameters formatted as a codestring or a block. A codestring is provided in the same way as a string parameter, with the exception that the string must be a valid Beatbox script *expression*. The expression should compile to the variable type specified in the device's documentation. Codestrings allow devices to evaluate an expression as the simulation is running, rather than reduce it to a literal value.

A block is a string enclosed in braces (`{ }`). The contents of a block will be passed literally to the device, without intervention from the script interpreter. This allows a block to contain characters such as `;` or `"`, that could not be included in a string parameter. Some devices, notably `k_func`, use a block to accept script statements. `euler` uses a block to pass parameter

assignments to its RHS module. Examples of both are shown below. Listing G.5 shows the assignment of a parameter, `bar` to the `foo` device. Although the value of `hat` may change throughout the simulation, the value of the `bar` parameter will be set only once, when the device is called. In this case, `bar` will be assigned the value `40`.

```
def real hat 10.0;
foo bar = min(hat,60)*4;
```

Listing G.5: Assigning a device parameter.

```
k_func nowhere=1 pgm={
    stim = eq(t,stimTime);
};

euler v0=0 v1=neqn-1 ht=ht ode=crn par={ht=ht, IV=@[iext]};
```

Listing G.6: Assigning block parameters to the `k_func` and `euler` devices.

G.5.2 Generic Device Parameters

A set of generic parameters, listed below, is applicable to all device types. All of the generic parameters listed below are optional — if no value is given for them, a default will be provided by Beatbox. Generic parameters and their defaults are described below.

`(str) name` Given name for the device. If the script specifies two instances of a device, giving one of the devices a `name` will disambiguate the devices in any output messages. Also, some devices refer to another device in the simulation using its `name` parameter.

`(real) when` The device's *condition*. The value given for `when` must be a variable of type `real`. A literal value cannot be used for this parameter. The device will be run only on timesteps when `when` evaluates to TRUE. For devices that should run on every timestep, Beatbox provides the system variable `when=always`, whose value is `1`. `always` is the default value for the `when` parameter. A pointer to the condition variable is assigned to the `c` field of the `Device` structure.

Space Parameters These specify the points of the mesh on which the device will operate:

`(int) x0` Lower x bound. Defaults to `X0`.

`(int) x1` Upper x bound. Defaults to `X1`.

`(int) y0` Lower y bound. Defaults to `Y0`.

`(int) y1` Upper y bound. Defaults to `Y1`.

`(int) z0` Lower z bound. Defaults to `Z0`.

`(int) z1` Upper z bound. Defaults to `Z1`.

`(int) v0` Lower v (dynamic variable) bound. Defaults to `0`.

`(int) v1` Upper v (dynamic variable) bound. Defaults to `v0`.

`(int) nowhere` Indicates that the device doesn't operate on the simulation medium. Defaults to `0`.

In general, a device will operate on points from, e.g., `x0` $\leq x \leq$ `x1`. In some cases, a device may use space parameters to specify specific points or dynamic variables, as opposed to ranges. For example, Beatbox's `diff*` devices use `v0` to indicate the dynamic variable containing transmembrane potential and `v1` to indicate where the Laplacian should be assigned. For devices that have no effect on the simulation medium, the `nowhere` parameter should be set to `1` to indicate this. The operation of some devices, in particular `k_func` is strongly affected by the `nowhere` parameter.

Window Parameters These specify the placement and colouring of device's on-screen displays.

`int row0` Lower row bound.

`int row1` Upper row bound.

`int col0` Lower column bound.

`int col1` Upper column bound.

`int color` Colour.

G.6 The `k_func` Device

The `k_func` device allows you to specify statements of Beatbox script code to be executed as part of the simulation. Script code run by a `k_func` device has access to variables of the simulation medium and to the k_variables you define in the script. The script code run by a `k_func` device, referred to as its *program*, is specified in the `pgm` parameter.

A common application of `k_func` devices is in updating variables used for devices' `when` parameters. For example, to run another device at timestep `200` only, the `k_func` device called below will assign the value `1.0` to `stim` when the current timestep, `t` equals `stimTime`:

```
def real stimTime 200;
k_func nowhere=1 pgm={
  stim = eq(t,stimTime);
};
```

The `pgm` parameter takes script code between curly braces ({ }). There may be multiple lines, each terminated by a semicolon. The following device call adds a line to the `pgm` parameter to set `print` to `1.0` on timesteps `0`, `50`, `100`, ...:

```
def real stimTime 200;
def real printInterval 50;
k_func nowhere=1 pgm={
  stim = eq(t,stimTime);
  print = ifeq0(mod(t,printInterval));
};
```

To change the values of k_variables in this way requires that the `k_func` device's `nowhere` parameter to be set to `1`. Alternatively, if the `k_func` device has a space, it can operate on the simulation medium. When run with a space (and `nowhere=0`), `k_func` iterates over its space, executing the program at each point, with data from that location of the simulation medium. Dynamic variables at that location are accessible using variables beginning with `u`. For example, `u3` is the fourth dynamic variable. The example below stimulates the simulation medium by raising the transmembrane voltage on one half of the simulation medium.

```
def str V 0;
k_func when=stim x1=50 pgm={
    u[V] = u[V] + 1.5;
};
```

G.6.1 Phase Distribution

k_func provides a phase distribution feature, which allows values from a file to be mapped across the simulation medium. The file to be used is specified to the file parameter. The file is assumed to contain real numbers in space-separated, tabular form. For example, the following data file contains a 2×10 table of values:

```
2.378   5.111
5.768   8.860
5.609   7.777
1.493   6.545
5.609   7.777
1.347   7.346
4.256   7.423
4.167   6.245
1.234   7.987
3.245   5.222
```

Listing G.7: Sample k_func file.

Conceptually, the values are arranged in a circle, as shown in Figure G.1.

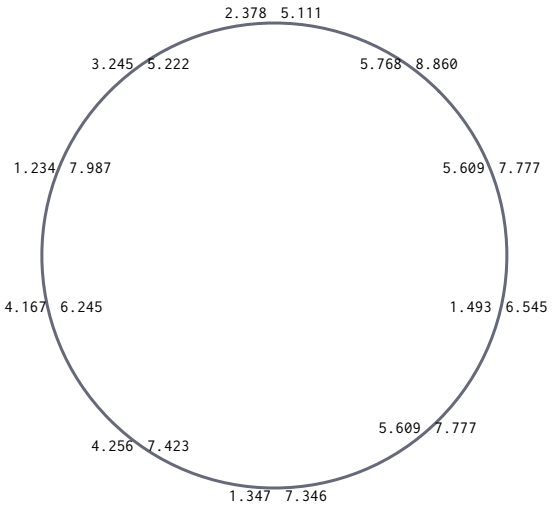


Figure G.1: Conceptual arrangement of file for use with phase distribution.

Phase distribution is initiated by assign a value to phaseu in the k_func program. The value assigned to phaseu, taken to be in radians, indicates a point on the circle, and consequently the line of the input file to be read. Since phaseu is a real number, linear interpolation is used to calculate values from two neighbouring rows of array, as illustrated in Figure G.2. The calculated value is then assigned to the corresponding layer of u.

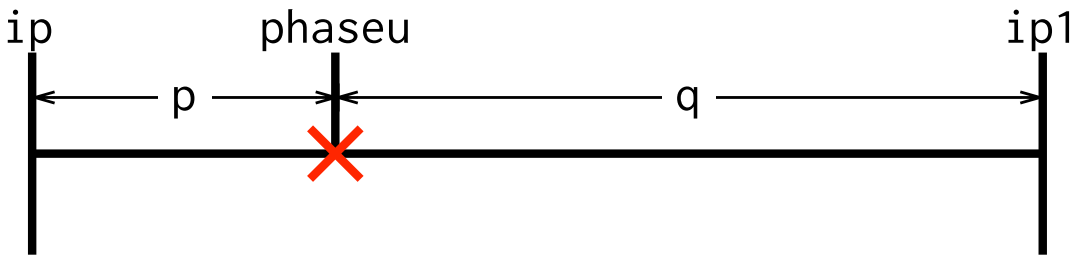


Figure G.2: `k_func` interpolation. The value of `phaseu` is interpolated between `ip` and `ip1` using `p` and `q`.

APPENDIX

H

BEATBOX DEVICE REFERENCE

H.1 `clock`

H.1.1 Overview

Prints the current timestep to `stdout`.

H.1.2 Parameters

None.

H.2 `ctlpoint`

H.2.1 Overview

Saves the entire simulation state for disaster recovery, or to facilitate a branching point in long simulations.

H.2.2 Parameters

Type	Name	Description
`str`	`file`	Control point file to be restored from / saved to.
`int`	`enrich`	Can the restored control point define k_variables not present in the script?

H.3 `diff`

H.3.1 Overview

Computes the Laplacian, with implicit Neumann boundary conditions.

H.3.2 Parameters

`diff` must be used with the default space. The `v0` space parameter is used to indicate the diffused variable, usually the transmembrane voltage. The `v1` space parameter is used to indicate the layer in which the Laplacian should be stored.

Type	Name	Description
Isotropic & Anisotropic		
`real`	`hx`	Space step.
Isotropic Only		
`real`	`D`	Isotropic diffusion coefficient.
Anisotropic Only		
`real`	`Dpar`	Anisotropic diffusion coefficient parallel to the fibres.
`real`	`Dtrans`	Anisotropic diffusion coefficient in the transverse direction, perpendicular to the fibres.

H.4 `dump`

H.4.1 Overview

Writes the contents of the simulation medium within its space to a binary file.

H.4.2 Parameters

Type	Name	Description
`str`	`file`	Relative path to the dump file.
`int`	`append`	Should the file be appended to?

H.5 `euler`

H.5.1 Overview

General pointwise ODE stepper.

H.5.2 Parameters

Type	Name	Description
`real`	`ht`	Timestep duration.
`str`	`ode`	Name of the RHS module to use.
`int`	`rest`	Number of timesteps of the RHS module to run before starting simulation. Can be useful for finding resting initial conditions.

H.6 k_func

H.6.1 Overview

Allows the user to specify user defined functions to be executed as part of the simulation run.

H.6.2 Parameters

Type	Name	Description
str	file	Relative path to file for phase distribution. See Beatbox Scripting Guide.
str	debug	Relative path to output file for debugging purposes. If "stdout", will print to screen.
Codeblock	pgm	The function to be executed when the device is run. Beatbox script statements, separated by semicolons.

Users should note that the behaviour of k_func changes significantly in relation to the nowhere parameter. When nowhere=1, assignments can be made to k_variables only. Assignments from the simulation medium or random number generators are disallowed. When nowhere=0, assignments can be made to the simulation medium only.

It is possible to define intermediate variables in k_func programs, using the def keyword. Variables must be initialised before they are used, and are reinitialised on each iteration of the k_func program.

H.7 load

H.7.1 Overview

Loads data from a file saved by dump into the simulation medium.

H.7.2 Parameters

Type	Name	Description
str	file	Relative path to the file to be read.
int	rewind	Should the file be rewound before reading?

H.8 poincare

H.8.1 Overview

Samples a dynamic variable at a single point in the mesh, to detect a crossing in time of a specified value.

H.8.2 Parameters

The `poincare` device must be run on a single point. Upper space parameters, `x1`, `y1`, `z1` and `v1` are unused by this device. The `v0` space parameter is the dynamic variable to be sampled at point (`x0`,`y0`,`z0`).

Type	Name	Description
k_variable	`result`	Name of the k_variable into which the result should be stored — `1` if a crossing has been detected on this timestep, `0` otherwise.
k_variable	`timestep`	Name of the k_variable into which the timestep of the last crossing should be stored. An assignment will only be made to this variable on the timestep when a crossing is detected.
`real`	`cross`	The crossing threshold. If previous and current values of the sampled dynamic variable straddle `cross`, a crossing is detected.
`int`	`sign`	The sign of the crossings to be detected. `1` indicates positive crossings only, `-1` indicates negative crossings only, `0` indicates that crossings in either direction will be accepted.

H.9 ppmout

H.9.1 Overview

Produces PPM images of the space. Values from specified layers of the simulation medium are mapped to colour channels by linear interpolation using specified minima and maxima. `ppmout` produces a new, sequentially numbered file every time it is run. In three-dimensional meshes, `ppmout` produces non-standard PPM files, in which multiple z layers are stacked into a single file.

H.9.2 Parameters

Type	Name	Description
Sequence	file	Format specifier for the sequence of files produced.
int	r	Layer of the simulation medium from which the red colour channel should be mapped.
real	r0	Value of the layer assigned to r that should map to 0.
real	r1	Value of the layer assigned to r that should map to 255.
int	g	Layer of the simulation medium from which the blue colour channel should be mapped.
real	g0	Value of the layer assigned to g that should map to 0.
real	g1	Value of the layer assigned to g that should map to 255.
int	b	Layer of the simulation medium from which the green colour channel should be mapped.
real	b0	Value of the layer assigned to b that should map to 0.
real	b1	Value of the layer assigned to b that should map to 255.
Geometry Only		
int	bgr	Red value of the background colour, to be displayed at void points.
int	bgg	Green value of the background colour, to be displayed at void points.
int	bgb	Blue value of the background colour, to be displayed at void points.

H.10 record

H.10.1 Overview

record writes values from the simulation medium to file in ASCII format. A hierarchical set of separators — vsep, xsep, ysep and zsep — are used between values. recordsep is used between records, i.e. separate runs of the record device.

H.10.2 Parameters

Type	Name	Description
str	file	Relative path to the file to be written.
int	append	Should the file be appended to?
str	filehead	Data to be placed at the top of the record file.
str	vsep	v-axis separator. Must be at most 2 characters in length.
str	xsep	v-axis separator. Must be at most 2 characters in length.
str	ysep	v-axis separator. Must be at most 2 characters in length.
str	zsep	v-axis separator. Must be at most 2 characters in length.
str	recordsep	Record separator. Must be at most 2 characters in length.

H.11 `reduce`

H.11.1 Overview

Reduce performs a reduction operation — sum, product, minimum or maximum — over its space. The result of the operation is assigned to a specified k_variable.

H.11.2 Parameters

Type	Name	Description
k_variable	`result`	Name of the k_variable into which the result should be stored.
`str`	`timestep`	Name of the operation to perform. Accepted values are "`sum`", "`product`", "`min`" and "`max`".

H.12 `sample`

H.12.1 Overview

Assigns a dynamic variable value from a specified point in the mesh to a k_variable.

H.12.2 Parameters

The `sample` device must be run on a single point. Upper space parameters, `x1`, `y1`, `z1` and `v1` are unused by this device. The `v0` space parameter is the dynamic variable to be sampled at point (`x0`,`y0`,`z0`).

Type	Name	Description
k_variable	`result`	Name of the k_variable into which the sampled value should be stored.

H.13 `stop`

H.13.1 Overview

Stops the simulation run.

H.13.2 Parameters

None.

APPENDIX

I

BEATBOX DEVELOPER GUIDE

I.1 Hello Beatbox!

Devices are the primary building block of a Beatbox simulation. A simulation consists of two or more devices that are called in turn, at most once for each simulation timestep. When called, the device can perform a task, with optional access to the simulation medium, `New`.

In this tutorial, we'll build a device to output the text "Hello, Beatbox!" in the simulation's standard output file. We won't go into much detail in this section, rather we'll just look at the absolute minimum one has to do to build a working device.

I.1.1 The Device Skeleton

The template below shows the a *.c* file containing the basic outline of a minimal device.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "system.h"
#include "beatbox.h"
#include "device.h"
#include "state.h"
#include "bikt.h"
#include "qpp.h"

typedef struct {
    int dummy;  // Add your own device-specific storage here.
} STR;

RUN_HEAD(myDevice)
/* Put code to perform the device's task here. */
RUN_TAIL(myDevice)

DESTROY_HEAD(myDevice)
/* Free any allocated resources here. */
DESTROY_TAIL(myDevice)
```

```
CREATE_HEAD(myDevice)
/* Add any initialisation code here. */
CREATE_TAIL(myDevice,1)
```

There are four key parts to a device. The first is the definition of the `STR` datatype. Every device in Beatbox defines `STR` to meet its specific storage requirements. Usually, `STR` is used to define a structure for any constants required by the device. For our "Hello, Beatbox!" device, we won't need any persistent storage, so we'll just use the minimal implementation that's given in the template. We'll see how to use `STR` in more detail in Section I.2.

The second part of interest is the function used to run our device. Code between the `RUN_HEAD` and `RUN_TAIL` macros defines what the device *does*. This is where whatever computation or output tasks to be performed by the device will happen. For our "Hello, Beatbox!" device, this is where we want to output our message. We'll use the `MESSAGE` macro, defined in *beatbox.h*, which will print to both `stdout` and the simulation's *.res* file.

```
RUN_HEAD(hello)
    MESSAGE("Hello, Beatbox!\n");
RUN_TAIL(hello)
```

We'll also have to change the name in the parentheses of the `RUN_HEAD` and `RUN_TAIL` macros from `myDevice` to `hello`. `hello` will be the name of our device, and we'll need to use it wherever we refer to the device. Device names cannot use spaces and are, by convention, all lower-case, without punctuation. If you really need to separate words, use an underscore (`_`).

The function defined by expanding the `DESTROY_HEAD` and `DESTROY_TAIL` macros is called only once, at the end of a simulation, and should be used to free any allocated resources. Since our device won't allocate anything, we don't have anything to free, so we can leave this function empty and just put the device name in the parentheses:

```
DESTROY_HEAD(hello)
/* Nothing to do here. */
DESTROY_TAIL(hello)
```

The fourth and final part prepares the device to be run. The function defined by expanding the `CREATE_HEAD` and `CREATE_TAIL` macros will be called only once, at the start of the simulation and should be used to perform any initialisation tasks required by the device. This is discussed in much more detail in Section I.2. For our "Hello, Beatbox!" device, we needn't add any code here, so just put the device name in the parentheses.

```
CREATE_HEAD(hello)
/* Nothing to do here. Quite boring, actually. */
CREATE_TAIL(hello,1)
```

Although we've not put any code between the `HEAD` and `TAIL` macros, it's important not to remove them altogether, as they are all referred to elsewhere in Beatbox. Also, be sure not to change the order of the macros, since `CREATE_TAIL` defines code dependent upon the `RUN_HEAD` and `DESTROY_HEAD` macros.

We can now save our device file. By convention, the device's *.c* file should have the same name as the device, so ours will be *hello.c*.

I.1.2 Registering our Device

Now that we have a new device, we need to inform Beatbox that it's available for use in a simulation. We do this by listing our device's name in *devlist.h*. The name listed here must match exactly (case matters) the name we used in our device code.

Open *devlist.h* and add the following line:

```
D(hello)
```

It's good practice to add your device in alphabetical order. You'll see a few devices in there that are listed as `S(somedevice)`. These devices will only work in sequential mode, and will be disabled in parallel. They're naughty devices. Since we don't want our new device to fall in with this rough crowd, we're going to use `D()`, *hmmkay*?

I.1.3 Building Beatbox with our Device

Beatbox is built using the GNU Autotools. To include our new device in the build, we must add it to *Makefile.am*, which, like your *hello.c* file, should be in the *src* directory. *Makefile.am* contains a simple description of what's required to build the software. When we're done changing it, we'll use `automake` to generate a new *Makefile*, from which the code will actually be built.

Open *Makefile.am*. Scroll down the file, past the flags and other stuff until you find the line

```
common_sources = \
```

followed by a long list of *.c* and *.h* files. In correct alphabetical order, add the following line to the list:

```
hello.c\
```

We can now build Beatbox with our new device. First, we need to generate a new *Makefile* by invoking `automake` from the *beatbox* directory.

```
$ pwd
$ /Users/rossmcf/Working/beatbox
$ automake
```

With `automake`, no news is good news. We can now compile our new version of Beatbox containing the `hello` device. We'll run the local *configure* script to set up the build system for our machine, and then `make` to actually build the code.

```
$ pwd
$ /Users/rossmcf/Working/beatbox
$ ./configure

... (Lots of stuff)

$ make

... (Lots more stuff, hopefully no errors.)
```

This will compile the code. Assuming all is well, you can now install the binary wherever you want it.

```
$ make install

... (Even more stuff.)
```

Don't be too worried about all the text whizzing by. If anything goes amiss, the last line usually tells you all you need to know.

I.1.4 Running our Device

To include our new device in a simulation, we use the device's name in the script. So, to print our "Hello, Beatbox!" at each timestep, we include the following line, making sure to leave a space between the device name and the semicolon.

```
hello ;
```

You can try using the following simple script, let's call it *basic.bbs*, to test it:

```
// The medium
state xmax=1 ymax=1 zmax=1 vmax=4;

// Schedule and display parameters
def real begin;
def real end;
k_func nowhere=1 pgm={
  begin =eq(t,0);
  end =ge(t,20);
};

// Our new device!
hello ;

stop when=end;
end;
```

Running the script with the following command:

```
$ Beatbox basic.bbs
```

should produce something like:

```
88888888ba                                  88
88      "8b                          ,d     88
88      ,8P                          88     88
88aaaaaa8P'   ,adPPYba,  ,adPPYYba,  MM88MMM  88,dPPYba,    ,adPPYba,  8b,     ,d8
88""""""8b,  a8P_____88  ""     'Y8    88     88P'    "8a  a8"     "8a  'Y8, ,8P'
88      '8b  8PP"""""""  ,adPPPPP88    88     88       d8  8b       d8    )888(
88      a8P  "8b,   ,aa  88,    ,88    88,    88b,   ,a8"  "8a,   ,a8"  ,d8" "8b,
88888888P"    '"Ybbd8"'  '"8bbdP"Y8    "Y888  8Y"Ybbd8"'    '"YbbdP"'  8P'     'Y8
--------------------------------------------------------------------------------
BEATBOX_1.0 -- MPI version date Oct 26 2010 22:11:47 $
execution begin at Tue Oct 26 22:14:12 2010 $
Input file basic.bbs without additional arguments $
with options: noappend autostop nodebug noverbose nograph noprofile resname=basic.res
state   Simulation medium (1 x 1 x 1)
    Domain decomposed as (001,001,001).
    0 processors of 1 remain idle.
$
k_func $
hello $
stop $
end of input file $
Loop of 3 devices created: (0)k_func (1)hello (2)stop $
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
Hello, Beatbox!
```

```
TIMING INFO
-------------------------
This run took 0.000266 seconds.


BEATBOX_0.1 finished at t=20 by device 2 "stop"
```

So, now that you've built your first device, you might want to make one that actually *does something*. For that, you'll need to read Section I.2.

I.2 Anatomy of a Device

This section assumes that you've read Section I.1 and have built a Beatbox device using the method described therein. We're now going to go a little more slowly, taking the time to see how and why things work they way they do in Beatbox.

Much of the generic code required of a device is provided as a collection of macros and type definitions in *device.h*, which must be included by all devices.

I.2.1 The Parameter Structure

For the persistent storage of data, devices must declare a *parameter structure* using the `STR` datatype. `STR` is a marker type that should be defined, using `typedef`, as a `struct`. Typically, the `STR` `struct` has fields for one or more user-specified parameters and any other constants required by an instance of the device. A minimal implementation, as shown in *hello.c*, above, should define `STR` as a `struct` with at least one field.

Commonly, the parameter structure is used to hold values of parameters read from the script. This process is described in Section I.2.5.

I.2.2 The `Device` Datatype

The `Device` datatype (*device.h*) represents an instance of a device, as called in the user script. One `Device` structure is allocated for each of the device calls in the script, such that if the same type of device (e.g. `k_func`) is called multiple times, there will be multiple `Device` structures. For the most part, the `Device` structure is only used 'behind the scenes'. One field of the `Device` structure that you will see a lot, however, is the `Space` structure. Your device's space determines the points and dynamic variables of the simulation medium on which it should operate.

A device's implementation code should not and interact directly with the `Device` structure, so we won't look at it any longer. Forget you even saw it. Some minimal interaction with the `Device` structure is provided in the function template macros defined in *device.h*. Using these is the best way to get the job done without wandering into any dangerous territory.

I.2.3 Device Function Templates

As we saw in Section I.1, your device needs to define three functions, referred to as Create, Run and Destroy. *device.h* defines three template macros that will wrap around your function implementations. Using these templates is strongly recommended since they ensure reliable operation of devices, keep code concise and reduce redundancy. Each template takes the form of a `HEAD` and `TAIL` macro. These open and close the function respectively, and throw in some standard functionality for good measure.

The head macros take at least one argument, the name of the device. This is included into the function's name when the macro is expanded, so that, for example, the `euler` device will

have functions called `create_euler()`, `run_euler()` and `destroy_euler()`. Your chosen device name must be unique and used consistently throughout all macros and in *devlist.h*.

`CREATE_HEAD` and `CREATE_TAIL`

You device's Create function performs any initialisation required for the device. The function is run once only, before the simulation begins. The `CREATE_HEAD` macro declares and allocates memory for the device's parameter structure, `S`, a local variable of type `STR`. You can use the body of the function to assign values to the parameter structure.

The defined function will have the name of `create_<name>`, where `<name>` is the argument to the `CREATE_HEAD` macro. To avoid name collisions, it is strongly recommended that `<name>` correspond to the name of the device. The `create_<name>` function has two parameters, `dev` and `w`. `dev` is a pointer to the `Device` structure and is used by the `CREATE_TAIL` macro to assign values to its fields. `w` is a pointer to the parameter string taken from the device call in the user script. This can be used in combination with the functions defined in *qpp.h* to acquire values from the user script. For more details on how to do this, see Section I.2.5.

The `CREATE_TAIL` macro assigns the address of `S` and pointers to functions named `run_<name>` and `destroy_<name>` to the `par`, `p` and `d` fields of the `Device` structure respectively.

Although the Create function has access to the `Device` structure at this point, it is perilous to tamper with the device structure, so just don't.

`RUN_HEAD` and `RUN_TAIL`

Your device's Run function describes the task it will perform. The function will be called `run_<name>` where `<name>` is the argument to the `RUN_HEAD` macro. The Run function is pointed to by the `p` field of the `Device` structure, with the `s`, `w` and `par` fields of the same structure passed as arguments. `s` is the device's `Space` structure, which is discussed in Section I.2.4. `w` is the device's `Window` structure. `par` is the device's parameter structure.

The `RUN_HEAD` macro contains code that assigns `par` to a local variable, `S`, of type `STR`. Fields of the parameter structure can then be accessed using `S->fieldName`.

Commonly, the Run function begins by retrieving values from the parameter structure to local variables. This can be done using shortcut macros defined in *device.h*. We discuss this in more detail in Section I.2.5.

The `RUN_TAIL` macro closes the function, returning `1` to indicate success. If the Run function of any device returns `0`, Beatbox will stop the simulation immediately.

`DESTROY_HEAD` and `DESTROY_TAIL`

In your device's Destroy function, it should put its toys away; it's time for bed. Specifically, the Destroy function should free any memory allocated for the device and close any open files. If you use the provided template macros, there is no need to free `par`/`S`. The function is pointed to by the `d` field of the `Device` structure, with the `s`, `w` and `par` fields of the same structure passed as arguments. `s` is the device's `Space` structure. `w` is the device's `Window` structure. `par` is the device's parameter structure.

The `DESTROY_HEAD` macro will again point local variable `S` to your device's parameter structure. The `DESTROY_TAIL` macro frees `par` and returns `1` to indicate success.

I.2.4 Acessing `New`

Beatbox's simulation medium is a four-dimensional data set — multiple dynamic variables or *layers* at points in three-dimensional space. Beatbox represents the simulation medium with a one-dimensional array called `New`, defined in *state.h*. In addition to exporting `New`, *state.h* provides a collection of variables and macros with which to navigate it. These are listed in Table I.1.

Name	Description
`t`	The current time step.
`New`	The simulation medium.
`vmax`	Number of layers.
`xmax`	Number of points in the x axis.
`ymax`	Number of points in the y axis.
`zmax`	Number of points in the z axis.
`vmax_xmax`	Number of variables in the v/x plane.
`vmax_xmax_ymax`	Number of variables in the $v/x/y$ plane.
`DX`	The number of array locations between neighbouring points on the x axis. (See Figure 2.8)
`DY`	The number of array locations between neighbouring points on the y axis. (See Figure 2.8
`DZ`	The number of array locations between neighbouring points on the z axis.
`DV`	The number of array locations between neighbouring points on the v (variable) axis.
`dim`	The dimensionality of the mesh, from `0` (single cell) to `3`.
`ONE`	`1` if the dimensionality is greater than or equal to 1. `0` otherwise.
`TWO`	`1` if the dimensionality is greater than or equal to 2. `0` otherwise.
`TRI`	`1` if the dimensionality is greater than or equal to 3. `0` otherwise.

Table I.1: Public `state` Variables

To access a point in `New`, use the `ind()` function to obtain the index of the desired location. `ind()` takes 4 coordinates, $\{x, y, z, v\}$. For a pointer, you will commonly use `New+ind(x,y,z,v)`, and for an absolute value `New[ind(x,y,z,v)]`.

When accessing `New`, your device MUST only alter values at points within its *space*, i.e. `i0 <= i <= i1`. When iterating over the medium, it is advised that the space structure's fields be used as loop bounds. Not doing so will produce inconsistent behaviour for users.

```
RUN_HEAD(example)
int x, y, z;
real HUGE *u;
for(z=s.z0;z<=s.z1;z++){
    for(y=s.y0;y<=s.y1;y++){
        for(x=s.x0;x<=s.x1;x++){
            u=New+ind(x,y,z,s.v0);
            /* do something to u here */
```

```
        }
    }
}
RUN_TAIL(example)
```

Accessing points immediately $(x \pm 1, y \pm 1, z \pm 1)$ outside of the device's *space* is acceptable, provided such access is *read-only*. If your device does reference neighbouring points in this fashion, it MUST include the `DEVICE_REQUIRES_SYNC` macro in its Create function. This will ensure that the device continues to operate correctly in parallel.

Great care should also be exercised when writing to locations in `New`. Since `New` may hold values relating to more than one timestep, the mathematical procedure may be corrupted if values required by other devices are overwritten. In general, operations on `New` MUST be commutative — the device MUST NOT assume a particular order of execution.

In addition, devices that are not explicitly parallelised MUST NOT assign values derived from `New` to their parameter structure, or any other global variable. This will stop terrible things from happening when the device is run in parallel.

To keep users happy, layers accessed by your device should be specified via the `Space` structure. In most cases, `v0` and `v1` describe a range, but some devices use them separately, e.g. `s.v0` as 'source' and `s.v1` as 'destination'.

Tissue Geometry

If your device is interested in accessing data about the underlying tissue geometry, you'll find it in `Geom`. `Geom` is organised in much the same way as `New`, and we use `geom_ind(x,y,z,v)` to access it. The `v` variable can have one of four values: `GEOM_STATUS`, `GEOM_FIBRE_1`, `GEOM_FIBRE_2` or `GEOM_FIBRE_3`. To test if a point is tissue, use `isTissue(x,y,z)`. The same rules apply to `Geom` as to `New`, in particular, stick to your `Space` structure and don't reference any further away than $(x \pm 1, y \pm 1, z \pm 1)$.

I.2.5 Reading Parameters

Commonly, a device will take some parameters from the user via the script. We call this process 'accepting' a parameter. There are two steps to this process: Firstly, the parameter is read by the script preprocessor, and assigned to the device's parameter structure. Later, in order to use the parameter, the device will retrieve the value from the parameter structure. Both of these steps are greatly simplified by some handy macros defined in *qpp.h*.

Parameter Accept Macros

Parameter accept macros provide a quick way to pull a parameter value from the script and assign it to its namesake field in the device's parameter structure. The first thing to consider is the type of parameter you'd like to accept. There are four common types:

Integers Requires an `int` in the parameter structure.

Real Numbers Requires a `double` in the parameter structure.

Strings Requires a `char[]` array in the parameter structure.

Files Requires two fields in the parameter structure; a `FILE *` with the same name as the parameter, and a `char[]` array of size `MAXPATH` with the name `<param>name`, where `<param>` is the name of the parameter.

The corresponding parameter accept macros are listed below. They each have a default argument that, if given the corresponding *null* value, will *require* that the user supplies a value, rather than merely making it optional.

```
ACCEPTI(name,deflt,minv,maxv);
```

`name` Name of the parameter whose value is to be read.

`deflt` Default value, in case no parameter called "name" is found. If `INONE`, a script parameter MUST be found.

`minv` Minimum allowed value. If `INONE`, no minimum is prescribed.

`maxv` Maximum allowed value. If `INONE`, no maximum is prescribed.

```
ACCEPTR(name,deflt,minv,maxv);
```

`name` Name of the parameter whose value is to be read.

`deflt` Default value, in case no parameter called "name" is found. If `RNONE`, a script parameter MUST be found.

`minv` Minimum allowed value. If `RNONE`, no minimum is prescribed.

`maxv` Maximum allowed value. If `RNONE`, no maximum is prescribed.

```
ACCEPTS(name,deflt);
```

`name` Name of the parameter whose value is to be read.

`deflt` Default value, in case no parameter called "name" is found. If `NULL`, a script parameter MUST be found.

```
ACCEPTF(name,mode,default);
```

`name` Name of the parameter whose value is to be read.

`mode` Access mode in which the file is to be opened. Syntax is identical to a `fopen()` call:

`r` **or** `rb` Open existing le for reading.

`w` **or** `wb` Create le or wipe existing le before writing.

`a` **or** `ab` Append to end of existing le, creating if necessary.

`rt` **or** `rbt` **or** `rtb` Open existing le for updatereading and writing.

`wt` **or** `wbt` **or** `wtb` Create le or wipe existing le before updating.

`at` **or** `abt` **or** `atb` AppendOpen or create le for update, writing at end of le.

`deflt` Default filename, in case no parameter called "name" is found. If `NULL`, a filename MUST be found in the script.

Parameter Shortcut Macros

To aid access to fields of the parameter structure, *device.h* provides shortcut macros to declare local variables and initialise them with values from their namesake fields in the parameter structure. `CONST` is used to copy values from `S` to a local variable. `VAR` copies the address of a variable in `S` to a local pointer. `ARRAY` copies the address of the first element of an array in `S` to a local pointer.

I.2.6 The Laws of Devices

Beatbox allows simulations to be run on distributed-memory parallel machines using MPI. Many devices can be run in parallel without special adaptation, but it's difficult to tell exactly what will and won't work in parallel. The 'Laws of Devices' below describe how a device should behave in order to ensure its safety in parallel. If these rules aren't followed, there is a very real chance that your device will crash horribly in parallel, or worse, quietly produce incorrect results. Please read and stick to them.

1. A device must only alter layers of `New` in its space, at points in its space.
2. All operations on `New` must be commutative — the order of operation cannot be assumed.
3. Aside from the permitted regions of `New`, a device may only alter the values of local variables or those in its parameter structure.
4. Assignments made to a device's parameter structure must not be derived from data in `New`.
5. **Assignments made to a device's parameter structure must not be derived from data in `New`.**
6. Assignments made to a device's parameter structure must not be derived from data in its `Space` structure.
7. Assignments made to a device's parameter structure must not be derived from local minima or maxima (e.g. `local_xmax`).
8. Assignments made to a device's parameter structure must not be derived from a random number generator.
9. From any point in its space, a device may not reference points in `New` beyond $\{x \pm 1, y \pm 1, z \pm 1\}$.
10. A device that references neighbouring points in `New` must put the `DEVICE_REQUIRES_SYNC` macro in its Create function.
11. **From any point in its space, a device may not reference points in `Geom` beyond $\{x \pm 1, y \pm 1, z \pm 1\}$.**
12. Files accessed from a device must only be read.

If you're planning on making a device that needs to do something forbidden by the laws above, it will need to be explicitly parallelised to run in parallel. Please consult your local MPI expert. In the meantime, you can enable the device as sequential only, by going to *devlist.h* and changing the `D` before your device's name to an `S`. It's now one of the naughty devices.

BIBLIOGRAPHY

D. Abramson, M. O. Bernabeu, B. Bethwaite, K. Burrage, A. Corrias, C. Enticott, S. Garic, D. Gavaghan, T. Peachey, J. Pitt-Francis, E. Pueyo, B. Rodriguez, A. Sher, and J. Tan. High-throughput cardiac science on the grid. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 368(1925):3907–3923, August 2010.

M. A. Allessie, F. I. M. Bonke, and F. J. G. Schopman. Circus movement in rabbit atrial muscle as a mechanism of tachycardia. *Circulation Research*, 33(1):54–62, July 1973.

Apple, Inc. Automator. URL `http://www.apple.com/automator`.

O. Aslanidi, A. Benson, R. Clayton, and G. Halley. The virtual ventricular wall: A tool for exploring cardiac propagation and arrhythmogenesis. *Journal of Biological Physics*, January 2006.

N. Ayache, H. Delingette, M. Sermesant, S. Gilbert, O. Bernus, A. Holden, and A. Benson. *A Quantitative Comparison of the Myocardial Fibre Orientation in the Rabbit as Determined by Histology and by Diffusion Tensor–MRI*, volume 5528, pages 49–57. Springer, Berlin / Heidelberg, 2009.

D. Barkley. A model for fast computer simulation of waves in excitable media. *Physica D: Nonlinear Phenomena*, 49(1-2):61–70, 1991.

K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

G. W. Beeler and H. Reuter. Reconstruction of the action potential of ventricular myocardial fibres. *The Journal of Physiology*, 268(1):177–210, June 1977.

M. O. Bernabeu, M. J. Bishop, J. Pitt-Francis, D. J. Gavaghan, V. Grau, and B. Rodríguez. High performance computer simulations for the study of biological function in 3D heart models incorporating fibre orientation and realistic geometry at para-cellular resolution. In *Computers in Cardiology*, pages 721–724, 2008.

O. Bernus, R. Wilders, C. Zemlin, and H. Verschelde. A computationally efficient electrophysiological model of human ventricular cells. *American Journal of Physiology- Heart and Circulatory Physiology*, January 2002.

V. Biktashev and A. Karpov. QUI software package. Unpublished.

V. N. Biktashev. Dissipation of the excitation wave fronts. *Physical Review Letters*, 89(16), September 2002.

I. V. Biktasheva, V. N. Biktashev, and A. V. Holden. Wavebreaks and self-termination of spiral waves in a model of human atrial tissue. *Functional Imaging and Modeling of the Heart*, 3504, 2005.

M. R. Boyett, A. Clough, J. Dekanski, and A. V. Holden. Modelling cardiac excitation and excitability. In A. V. Panfilov and A. V. Holden, editors, *Computational Biology of the Heart*, chapter 1. John Wiley & Sons Ltd., Chichester, 1997.

M. Buist, G. Sands, P. Hunter, and A. Pullan. A deformable finite element derived finite difference method for cardiac activation problems. *Annals of Biomedical Engineering*, 31(5):577–588, May 2003.

I. Bush, I. Todorov, and W. Smith. Optimisation of the input and output (I/O) in DL_POLY_3, 2010. URL `http://www.hector.ac.uk/cse/distributedcse/reports/DL_POLY01/DL_POLY01/index.htm`.

T. D. Butters, O. V. Aslanidi, S. Inada, M. R. Boyett, J. C. Hancox, M. Lei, and H. Zhang. Mechanistic links between Na^+ channel (SCN5A) mutations and impaired cardiac pacemaking in sick sinus syndrome. *Circulation Research*, 107(1):126–137, July 2010.

C. Clancy and Y. Rudy. Na^+ channel mutation that causes both brugada and Long-QT syndrome phenotypes a simulation study of mechanisms. *Circulation*, January 2002.

R. H. Clayton and A. V. Holden. Computational framework for simulating the mechanisms and ECG of re-entrant ventricular fibrillation. *Physiological Measurement*, 23(4):707, 2002.

R. H. Clayton and A. V. Panfilov. A guide to modelling cardiac electrical activity in anatomically detailed ventricles. *Progress in Biophysics & Molecular Biology*, 96(1-3):19–43, January 2008.

R. H. Clayton, C. P. Bradley, M. P. Nash, A. Mourad, M. Hayward, P. David, and T. Peter. Abstract 2351: Transient type II ventricular fibrillation during global ischemia in the in-situ human heart. *Circulation*, 120(18_MeetingAbstracts):S630, November 2009.

M. Courtemanche, R. J. Ramirez, and S. Nattel. Ionic mechanisms underlying human atrial action potential properties: insights from a mathematical model. *Am J Physiol Heart Circ Physiol*, 275(1):H301–321, 1998.

J. Crank and P. Nicolson. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Advances in Computational Mathematics*, 6(1):207–226, December 1996.

J. M. Davidenko, A. V. Pertsov, R. Salomonsz, W. Baxter, and J. Jalife. Stationary and drifting spiral waves of excitation in isolated cardiac muscle. *Nature*, 355(6358):349–351, January 1992.

D. Dubin. *Ion Adventure in the Heartland*, volume 1. 2003.

M. A. Dumett and J. P. Keener. An immersed interface method for solving anisotropic elliptic boundary value problems in three dimensions. *SIAM Journal on Scientific Computing*, 25(1): 348–367, 2003.

F. Fenton and A. Karma. Vortex dynamics in three-dimensional continuous myocardium with fiber rotation: Filament instability and fibrillation. *Chaos*, 8(1):20–47, March 1998.

F. H. Fenton, E. M. Cherry, A. Karma, and W.-J. Rappel. Modeling wave propagation in realistic heart geometries using the phase-field method. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 15(1):013502, 2005.

R. FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal*, January 1961.

P. Franzone and L. F. Pavarino. A parallel solver for reaction-diffusion systems in computational electrocardiology. *Mathematical Models and Methods in Applied Sciences*, January 2004.

P. C. Franzone, L. F. Pavarino, and B. Taccardi. Effects of transmural electrical heterogeneities and electrotonic interactions on the dispersion of cardiac repolarization and action potential duration: A simulation study. *Mathematical biosciences*, 204(1):132–65, November 2006.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of reusable object-oriented software*. 1995.

R. A. Gray, W. T. Baxter, C. Cabo, J. M. Davidenko, A. Pertsov, and J. Jalife. Video imaging of re-entry on the epicardial surface of the isolated rabbit heart. In A. V. Panfilov and A. V. Holden, editors, *Computational Biology of the Heart*, chapter 10. John Wiley & Sons Ltd., 1997.

R. A. Gray, A. M. Pertsov, and J. Jalife. Spatial and temporal organization during cardiac fibrillation. *Nature*, 392(6671):75–78, March 1998.

R. M. Gulrajani. The forward and inverse problems of electrocardiography. *EEE Engineering in Medicine and Biology Magazine*, 17(5):84–101,122, September–October 1998.

C. Henriquez and A. Papazoglou. Using computer models to understand the roles of tissue structureand membrane dynamics in arrhythmogenesis. *Proceedings of the IEEE*, 84(3), March 1996.

A. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, January 1952.

D. Hooks, K. Tomlinson, and S. Marsden. Cardiac microstructure: implications for electrical propagation and defibrillation in the heart. *Circulation Research*, January 2002.

P. Hunter, W. Li, A. McCulloch, and D. Noble. Multiscale modeling: Physiome project standards, tools, and databases. *Computer*, January 2006.

V. Iyer, R. Mazhari, and R. L. Winslow. A computational model of the human left-ventricular epicardial myocyte. *Biophys J*, 87(3):1507–25, September 2004.

A. M. Katz. *Physiology of the Heart*. Lippincott Williams & Wilkins, Philadelphia, 4th edition, 2006.

J. P. Keener and K. Bogar. A numerical method for the solution of the bidomain equations in cardiac tissue. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 8(1):234–241, March 1998.

B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, March 1988.

I. J. LeGrice. The architecture of the heart: a data-based model. *Philosophical Transactions: Mathematical*, January 2001.

C.-H. Luo and Y. Rudy. A dynamic model of the cardiac ventricular action potential. I. simulations of ionic currents and concentration changes. *Circulation Research*, 74(6):1071–1096, June 1994a.

C.-H. Luo and Y. Rudy. A dynamic model of the cardiac ventricular action potential. II. afterdepolarizations, triggered activity, and potentiation. *Circulation Research*, January 1994b.

A. McCulloch. Continuity 6 — a problem solving environment for multi-scale biology, May 2010. URL `http://www.continuity.ucsd.edu`.

Message Passing Interface Forum. MPI: A message-passing interface standard. November 2003.

G. R. Mines. On dynamic equilibrium in the heart. *Journal of Physiology*, 46(4–5):349–383, July 1913.

S. W. Morgan. *Low-Energy Defibrillation Using Resonant Drift Pacing*. PhD thesis, University of Liverpool, February 2009.

R. M^{c}Farlane. Integration of the luo-rudy(1994) model of ventricular action potential into the QUI package for the simulation of cardiac arrhythmias. Master's thesis, University of Liverpool, September 2007.

M. P. Nash, A. Mourad, R. H. Clayton, P. M. Sutton, C. P. Bradley, M. Hayward, D. J. Paterson, and P. Taggart. Evidence for multiple mechanisms in human ventricular fibrillation. *Circulation*, 114(6):536–542, August 2006.

P. Nielsen, I. L. Grice, B. Smaill, and P. Hunter. Mathematical model of geometry and fibrous structure of the heart. *American Journal of Physiology- Heart and Circulatory Physiology*, January 1991.

D. Noble. A modification of the Hodgkin–Huxley equations applicable to Purkinje fibre action and pacemaker potentials. *J Physiol*, January 1962.

D. Noble. Modeling the heart — from genes to cells to the whole organ. *Science*, January 2002. URL `http://www.sciencemag.org/cgi/content/abstract/295/5560/1678`.

D. Noble. Modelling the heart. *Physiology*, 19:191–197, 2004.

B. J. Noye and R. J. Arnold. Accurate finite difference approximations for the Neumann condition on a curved boundary. *Applied Mathematical Modelling*, 14(1):2–13, 1990. ISSN 0307-904X. doi: DOI: 10.1016/0307-904X(90)90157-Z.

A. V. Panfilov and A. V. Holden. *Computational Biology of the Heart*. 1997.

J. Pitt-Francis, P. Pathmanathan, M. O. Bernabeu, R. Bordas, J. Cooper, A. G. Fletcher, G. R. Mirams, P. Murray, J. M. Osborne, A. Walter, S. J. Chapman, A. Garny, I. M. M. van Leeuwen, P. K. Maini, B. RodrÌguez, S. L. Waters, J. P. Whiteley, H. M. Byrne, and D. J. Gavaghan. Chaste: A test-driven approach to software development for biological modelling. *Computer Physics Communications*, 180(12):2452–2471, 12 2009.

G. Plank, N. Trayanova, and K. Roy. Computational support systems for guiding delivery of cardiac therapies. Technical report, Cray Centre of Excellence, July 2010.

Python Software Foundation. Python programming language. URL `http://www.python.org`.

P. Ridley. Guide to partitioning unstructured meshes for parallel computing. Technical report, Numerical Algorithms Group, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UK, April 2010.

J. M. Rogers and A. D. McCulloch. A collocation–galerkin finite element model of cardiac action potential propagation. *IEEE Trans Biomed Eng*, 41(8):743–757, August 1994.

Y. Rudy. From genome to physiome: Integrative models of cardiac excitation. *Annals of Biomedical Engineering*, 28(8):945–950, 2000.

D. F. Scollan, A. Holmes, R. Winslow, and J. Forder. Histological validation of myocardial microstructure obtained from diffusion tensor magnetic resonance imaging. *AJP - Heart and Circulatory Physiology*, 275(6):H2308–2318, December 1998.

D. F. Scollan, A. Holmes, J. Zhang, and R. L. Winslow. Reconstruction of cardiac ventricular geometry and fiber orientation using magnetic resonance imaging. *Annals of Biomedical Engineering*, 28(8):934–944, August 2000.

M. Sermesant, P. Moireau, O. Camara, J. Saintemarie, R. Andriantsimiavona, R. Cimrman, D. Hill, D. Chapelle, and R. Razavi. Cardiac function estimation from mri using a heart model and data assimilation: Advances and difficulties. *Medical Image Analysis*, 10(4):642–656, August 2006.

L. A. Smith. Mixed mode MPI / OpenMP programming. Technical report, Edinburgh Parallel Computing Centre, 2000.

J. Sundnes, G. T. Lines, X. Cai, B. rn Fredrik Nielsen, K.-A. Mardal, and A. Tveito. *Computing the Electrical Activity in the Heart*. Monographs in Computational Science and Engineering. Springer, Berlin, 2006.

R. Sunthareswaran. *Cardiovascular System*. Mosby's Crash Course. 1998.

The PHP Group. PHP: Hypertext Preprocessor. URL `http://www.php.net`.

L. Tung. *A Bidomain Model for Describing Ischemic Myocardial D-C Potentials*. PhD thesis, Massachusetts Institute of Technology, Cambridge, June 1978.

K. T. Tusscher and A. V. Panfilov. Reentry in heterogeneous cardiac tissue described by the luo-rudy ventricular action potential model. *American Journal of Physiology- Heart and Circulatory Physiology*, January 2003.

K. T. Tusscher and A. V. Panfilov. Cell model for efficient simulation of wave propagation in human ventricular tissue under normal and pathological conditions. *Phys. Med. Biol*, January 2006.

K. T. Tusscher, O. Bernus, R. Hren, and A. V. Panfilov. Comparison of electrophysiological models for human ventricular cells and tissues. *Progress in Biophysics & Molecular Biology*, January 2006.

University of Oxford Computing Laboratory. Chaste, 2010. URL `http://www.comlab.ox.ac.uk/chaste`.

UoE HPCX Ltd. Hector hardware, 2007–2010. URL http://www.hector.ac.uk/service/hardware/.

F. van Capelle and D. Durrer. Computer simulation of arrhythmias in a network of coupled excitable elements. *Circulation Research*, 47(3):454–466, 9 1980. URL http://circres.ahajournals.org/cgi/content/abstract/47/3/454.

M. Veldkamp, P. Viswanathan, and C. Bezzina. Two distinct congenital arrhythmias evoked by a multidysfunctional Na^+ channel. *Circulation Research*, Jan 2000.

F. J. Vetter and A. D. McCulloch. Three-dimensional analysis of regional cardiac function: a model of rabbit ventricular anatomy. *Progress in Biophysics and Molecular Biology*, 69(2-3): 157–183, May 1998.

E. Vigmond, M. Hughes, G. Plank, and L. Leon. Computational tools for modeling electrical activity in cardiac tissue. *Journal of Electrocardiology*, 36(Supplement 1):69–74, 2003.

H. Watanabe, T. Koopmann, S. L. Scouarnec, and T. Yang. Sodium channel $\beta 1$ subunit mutations associated with Brugada syndrome and cardiac conduction disease in humans. *The Journal of Clinical Investigation*, 118:2260–2268, 2008.

N. Wiener and A. Rosenbleuth. The mathematical formulation of the problem of conduction of impulses in a network of connected excitable elements, specifically in cardiac muscle. *Arch. Inst. Cardiol. Mex.*, 16(205):3–4, 1946.

A. Wilde and C. Bezzina. Genetics of cardiac arrhythmias. *British Medical Journal*, Jan 2005.

R. Wolfe. OpenMP parallelisation of QUI. Technical report, University of Liverpool, Unpublished.