Extending the Gotran framework: LaTeX and GPU acceleration

George Bahij

University of Oslo/Simula Research Laboratory

[simula . research laboratory]

Abstract

Gotran *provides a framework for working with systems of ordinary differential equations (*ODEs*). Its primary goal is to increase the workflow efficiency of computational modelling in biomedical research. The ODEs, given by the time derivative of state variables, are described in a Gotran form file and can be automatically translated into different outputs depending on the user's needs. As of this writing, Gotran supports Python, MATLAB and C/C++ outputs. In this thesis we present extensions to Gotran and their implementations, including automatic generation of LaTeX output and GPU acceleration on Nvidia graphics cards.*

Contents

Part I
Background

1. Motivation

Recent advances in biomedical research over the last century have significantly contributed to the increased understanding and effectiveness in both combatting communicable diseases and treating non-communicable ones. With diseases of the heart remaining among the leading causes of death in both developed and developing countries [Murray and Lopez, 1997], a great deal of effort is placed into furthering our understanding of the heart and the nature of these diseases.

Modern research in cellular electrophysiology commonly employs mathematical models describing the electrical activity in cellular membranes. These are dynamic systems described by means of systems of ordinary differential equations (ODEs) and partial differential equations (PDEs). The models are useful in basic scientific research of the chemical, physical and electrical properties of living organisms and their biological constituents, such as the human heart. The results from this type of research are useful for further applications in clinical diagnostics and therapeutics, such as the efforts to undestand the underlying causes of and to develop potential treatments and preventative measures against cardiac disease.

The ODEs required to create accurate mathematical models of biological systems tend to be highly complex. As an example, consider the Winslow model [Winslow et al., 1999], which describes the electrochemical reactions of a single canine heart cell with a system of 31 ODEs. Non-linear ODE systems of such complexity have no known analytical solutions. Researchers must thus rely on computationally heavy numerical simulations of these models to generate meaningful results. "The cost of doing one single call to the Winslow ODE system is so big that a measure of efficiency is almost entirely based on the number of such calls" [Kaarby, 2007].

Subsequently, the mathematical models are tedious to develop, solve, share and publish. This leads to problems with reproducibility, and they are prone to typographical errors during transcription from simulation code to

its corresponding research manuscript. A specialised framework, *Gotran*, is being developed at the Simula Research Laboratory to aid researchers in working with ODEs more efficiently.

Extension and improvement of the Gotran framework is the core focus of this thesis.

1.1 Thesis outline

Sections 2, 3 and 4 provide an overview of background topics relevant to the framework at the core of this thesis. Section 5 introduces the Gotran framework.

Section 6 outlines the implementation requirements for the extensions to Gotran developed in this thesis. Sections 7 and 8 further describe the specific implementation details of the LaTeX output generation and GPU acceleration, respectively.

Section 9 presents and discusses test results and benchmarks for the GPU acceleration.

2. Cardiac modelling

The exponential increase in computational power of the past decades has opened up new possibilities in biomedical research. Mathematical models of increasing precision and complexity are continuously developed to better explain the properties and interactions of cardiac cells.

2.1 Action potential

Excitable cells, chiefly neurons and muscle cells, are those that produce a small electric current when stimulated in a brief event called an *action potential*. During such events, an electrical current crosses the cell membrane. Upon reaching a critical threshold, the stimulus triggers a runaway condition that rapidly depolarises the cell's membrane potential before quickly repolarising back to its resting potential. An action potential is followed by a refractory period where the cell is no longer excitable. The duration of the full cycle of an action potential varies significantly, from less than a millisecond in cerebral neurons to hundreds of milliseconds for the contraction of cardiac muscle cells.

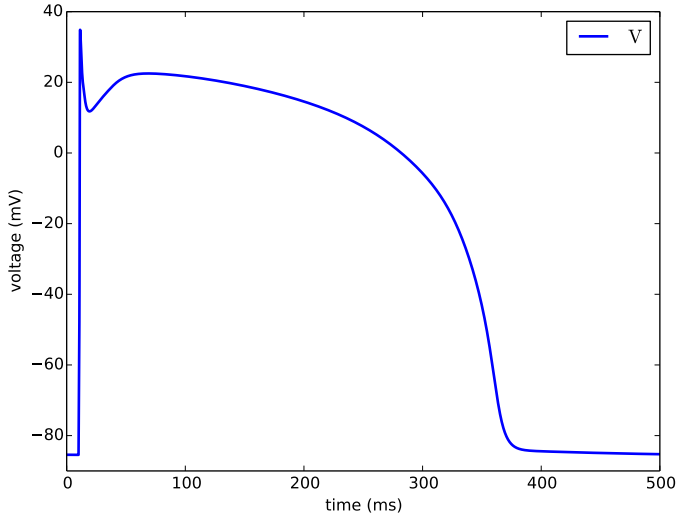


Figure 2.1: *Action potential in the human heart with the Ten Tusscher-Panfilov model*

The groundbreaking Hodgkin-Huxley model [Hodgkin and Huxley, 1952], first used to describe the mechanisms of the propagation of electrical signals in a squid giant axon by modelling each cellular component as an electrical element, forms the basis for the modern cell models used in this thesis. These cell models are described in Section 2.3.

Figure 2.1 illustrates a simulation of the action potential in the heart with the Ten Tusscher-Panfilov model [Ten Tusscher and Panfilov, 2006] of a human ventricular action potential.

2.2 Monodomain equation

At the cellular level, the travelling action potential activates ionic currents across the cell membrane that sustain the propagation. Mathematically, this phenomenon can be described by a reaction diffusion equation. In cardiac modelling, one such equation is described by the monodomain equation [Sundnes et al., 2006a,b]:

$$\chi \left(C_m \frac{\partial u}{\partial t} + I_{ion}(u, s) \right) - \nabla \sigma \nabla u = I_{stim} \tag{2.1}$$

$$\sigma \nabla u \cdot \mathbf{n} = 0 \; on \; \partial\Omega \tag{2.2}$$

$$\frac{\partial s}{\partial t} = f(u, s) \tag{2.3}$$

Here, u is the membrane potential, I_{ion} the ionic currents, s the state variables controlling the ionic currents, f a function describing the time derivative of s, σ the conductancy of the tissue, χ the ratio between the volume and surface area of a heart cell, and C_m the capacitance of the cell membrane. The monodomain equation is a coupled system of one PDE, Equations 2.1 and 2.2, and a system of ODEs, Equation 2.3. The latter is often called the cell model and describes the intricate dynamics between the different ionic currents and the membrane potential.

Discretising the monodomain equation using an implicit method creates huge linear systems to be solved for each simulated time step. For instance, to simulate the travelling action potential in a human heart at a partial resolution of 0.5 mm, one would need a mesh with about one million nodes. Using a standard finite element method, the number of degrees of freedom in our system would then be $N_{nodes} \cdot (N_s + 1)$, where N_s is the length of s. Typically, cell models contain 10-50 state variables [Beeler and Reuter, 1977; Grandi et al., 2010; Ten Tusscher and Panfilov, 2006], but can easily be even larger [Flaim and McCulloch, 2007], which in our example would result in systems with tens of millions of unknowns. The ODEs from the cell models are also non-linear, stiff problems, making the solving procedure harder still.

To simplify the problem, an operator splitting method is commonly employed [Sundnes et al., 2006a], where the reaction is split from the diffusion:

$$\frac{\partial u}{\partial t} - \nabla\sigma/\chi\nabla u = I_{stim}/\chi \tag{2.4}$$

$$\sigma\nabla u \cdot \mathbf{n} = 0 \; on \; \partial\Omega \tag{2.5}$$

$$C_m\frac{\partial u^*}{\partial t} = -I_{ion}(u^*, s); \quad \frac{\partial s}{\partial t} = f(u^*, s) \tag{2.6}$$

Here, Equations 2.4 and 2.5 describe a clean diffusion equation with only one variable u, while Equation 2.6 describes the reaction dynamics at the cellular level. The solution u from the diffusion step is used as the initial condition for the ODE step, and the solution of the membrane potential from the ODE step, u^* is used as the initial condition for the PDE step. The total number of states for which a solution is calculated in Equation 2.6 is still $N_{nodes} \cdot (N_s + 1)$, but now the states at each node are decoupled from each other, letting us solve each node in parallel.

2.3 Cell models

The cell models used in this thesis are primarily concerned with modelling the membrane voltage of cardiac cells. There is a broad range of such models beyond the scope of this thesis. This section briefly introduces the three models that will be used in later sections for testing. These models rely on the assumption made by Hodgkin and Huxley [Hodgkin and Huxley, 1952] that cellular membranes act as capacitors, allowing a charge imbalance to form across the membrane.

Beeler-Reuter: A generic mathematical model of the membrane action potentials of ventricular myocardial fibres in mammals [Beeler and Reuter, 1977]. It employs a system of eight ODEs, four of which represent components of ionic current described by Hodgkin-Huxley-type equations.

Ten Tusscher-Panfilov: A human ventricular cell model developed to study "the conditions for alternans and spiral breakup in human cardiac tissue" [Ten Tusscher and Panfilov, 2006]. It consists of a system of 19 state variables.

Grandi-Pasqualini-Bers: A recent model from 2009 describing the human ventricle [Grandi et al., 2010] by using a system of 42 ODEs with independent state variables. This model is also stiffer than the others presented, making it more computationally expensive to calculate a stable numeric solution.

3. Ordinary differential equations

An ordinary differential equation (ODE) is a mathematical equation for an unknown function with one real independent variable that describes the relationship between the function's values and its derivatives of various orders. Systems of ODEs may contain several equations and state variables, but will still only include derivatives of one variable. These are separate from partial differential equations (PDEs) which involve functions of multiple independent variables and their corresponding partial derivatives.

The general nth-order implicit ordinary differential equation f can be described as

$$f\left(t,\ y(t),\ y\prime(t),\ y\prime\prime(t),\ \cdots,\ y^{(n)}(t)\right) = 0 \tag{3.1}$$

where $y : \mathbb{R} \to \mathbb{R},\ t \in \mathbb{R}$.

Solving a differential equation involves finding an exact or an approximate solution for y. Ordinary differential equations with non-linear terms or a high order typically do not have a known exact solution. An approximate solution for such ODEs is found through numerical computation and analysis.

3.1 Numerical methods for initial value problems

Two basic, widely used methods for such computations are the forward and backward Euler methods. They are described in Sections 3.1.1 and 3.1.2. Since these methods serve as important building blocks for more sophisticated methods, they have been tested as a part of the GPU-optimisation in this thesis, along with other methods as described in subsequent sections.

In this thesis we only consider first order ODEs, $y\prime(t) = f\,(t,\ y(t))$. The function f is also commonly known as the right hand side (rhs).

3.1.1 Forward Euler method

The forward or *explicit* Euler method is a basic method for solving ODEs numerically. As an explicit method, it only involves the current state of the system when calculating the next iteration.

Consider the first-order ordinary differential equation $y\prime(t) = f\,(t,\ y(t))$. Given an initial value $y(t_0) = y_0$ and a step size $h > 0$ for the independent variable t such that $t_n = t_0 + nh$ at the nth iteration, we can perform each successive iteration of the forward Euler method as follows:

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{3.2}$$

This gives us the approximation $y(t_n) \approx y_n$.

The approximation error for the forward Euler method grows proportionally to the step size for each iteration, both locally at each time step and globally over time. Thus, this method is unsuitable for approximating solutions to complex ODEs, especially stiff systemswhere it quickly becomes unstable without a very small step size.

3.1.2 Backward Euler method

The backward or *implicit* Euler method [Butcher, 1987] is similarly a basic method for numerically approximating ODE solutions. This method is

implicit, finding solutions using both the current and the successive state. The implicit Euler method is stable [Torelli, 1989] and thus far more suitable for stiff systems than the explicit counterpart.

As with the forward Euler method, consider a first-order ordinary differential equation of the form $y\prime(t) = f\,(t,\ y(t))$ with an initial value $y(t_0) = y_0$, and a step size h. The nth iteration can be calculated as follows:

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \tag{3.3}$$

3.1.3 Backward Euler with fixed-point iteration

As shown in Equation 3.3, the backward Euler method uses y_{n+1} in the right-hand side as opposed to the explicit method's y_n. Since y_{n+1} is used on both sides of the equation, an approximation must be calculated. One method is through fixed-point iteration as follows:

$$\begin{aligned} y_{n+1}^{[0]} &= y_n \\ y_{n+1}^{[i+1]} &= y_n + hf(t_{n+1}, y_{n+1}^{[i]}) \end{aligned} \tag{3.4}$$

Iterating this process until $|y_{n+1}^{[i+1]} - y_{n+1}^{[i]}| < \epsilon$ gives us an approximation $y(t_n) \approx y_n$.

3.1.4 Backward Euler with simplified Newton's method

An alternative to fixed-point iteration, as it might converge slowly, is the Newton's method. With Newton's method, given an initial approximation x_0, we may approximate the next solution x_1 as follows:

$$x_1 = x_0 - \frac{F(t_0, x_0)}{F'(t_0, x_0)} \tag{3.5}$$

Note that if we solve a system of ordinary differential equations, F' becomes a matrix. Let $y_{n+1} = x_0,\ \hat{y}_{n+1} = x_1$. Recalling the backward Euler method as $y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$, we may define F as shown:

$$\begin{aligned} F(t_{n+1}, y_{n+1}) &= y_{n+1} - y_n - hf(t_{n+1}, y_{n+1}) \\ \Rightarrow F'(t_{n+1}, y_{n+1}) &= \frac{dF}{dy_{n+1}} = 1 - hf'(t_{n+1}, y_{n+1}) \end{aligned} \tag{3.6}$$

Combining this back into Equation 3.5, we reach the following equation upon which we may iterate to converge towards a more accurate solution:

$$\begin{aligned} \hat{y}_{n+1} &= y_{n+1} - \frac{F(t_{n+1}, y_{n+1})}{F'(t_{n+1}, y_{n+1})} \\ \Leftrightarrow \hat{y}_{n+1} &= y_{n+1} - \frac{y_{n+1} - y_n - hf(t_{n+1}, y_{n+1})}{1 - hf'(t_{n+1}, y_{n+1})} \end{aligned} \tag{3.7}$$

In the case of complex cell models we solve a system of ordinary differential equations and f' becomes a matrix. We then need to solve a linear system for each iteration, which can be very time consuming for large systems. Following previous work [Lionetti, 2010, pp. 26—27], we will be using a single iteration of Newton's method with only the diagonal components of f', hereby referred to as the simplified backward Euler method, which yields the following equation:

$$y_{n+1} = y_n + \frac{hf(t_n, y_n)}{1 - hf'(t_n, y_n)} \tag{3.8}$$

The simplified backward Euler method is an explicit method and is therefore not as stable as the full method, but it is more stable than the explicit Euler.

3.1.5 Rush-Larsen methods

This thesis will also utilise the Rush-Larsen method [Rush and Larsen, 1978] for simulations in later sections. Originally formulated by Rush and Larsen in 1978 and demonstrated on the McAllister-Noble-Tsien [McAllister et al., 1975] cell model, this method was proposed as a computationally efficient alternative to the regular forward Euler method for ODE systems of first-order complexity. By recognising that most ODE systems describing cell models are quasi-linear, it modifies the steps of the forward Euler method with an analytical solution of the linear ODEs, while applying the standard forward Euler method to non-linear terms [Sundnes et al., 2009].

We will also use a slightly more complex extension of the Rush-Larsen method, detailed in [Sundnes et al., 2009]. By linearizing the non-linear terms, in addition to the linear, an analytical solution for all terms are used. This method will be referred to as the generalised Rush-Larsen method in this thesis.

3.2 Stiffness of cell model ODEs

The systems of ordinary differential equations describing cell models tend to bring some challenges for computing numerical solutions.

One such challenge arises from the *stiffness* of the ODE systems. An ODE system is stiff if it contains terms describing slow movements that are easily perturbed by nearby solutions that vary rapidly, leading to numerical instability when solved with certain numerical methods [Hairer and Wanner, 1999]. For stiff problems, such numerical methods require an extremely small time step to arrive at a stable solution and yield a useful result.

Simple solver algorithms such as explicit Euler are especially prone to instability problems caused by system stiffness. Combatting stiffness tends to involve self-correcting mechanisms during numerical computation. Methods implementing such mechanisms may produce stable results with much higher time steps at the expense of implementation complexity and per-iteration performance.

4. Graphics processing units and GPGPU

Graphics processing units (GPUs) are highly parallelised processors specialising in efficiently and repeatedly performing the same operations on large batches of data. This is primarily used for graphics processing on computationally heavy tasks such as 3D-rendering, texture mapping and geometric calculations to significantly ease the load on the central processing unit (CPU).

Despite recent years' advances in multithreaded parallelism on multiple cores, the architecture of a CPU is still designed primarily for fast serial processing of general-purpose computations. The GPU architecture is fundamentally different in this aspect, aimed at accelerating highly specific and massively parallelisable tasks with hundreds of cores in modern GPUs.

General-purpose computing on graphics processing units (GPGPU) is the repurposing of GPUs to perform computations for non-graphics related applications that have traditionally been handled by the CPU. This paradigm has seen a surge of popularity in data mining and computational science in recent years [Luebke et al., 2006]. With the increasing usage, several general-purpose programming languages and frameworks for high-

performance computing on GPU have been developed, including OpenCL[1] by Apple Inc. and Khronos Group, CUDA [NVIDIA Corporation, nd] by Nvidia and several others [Membarth et al., 2011].

4.1 GPGPU programming with CUDA

CUDA[2] is a parallel computing platform and programming model developed by Nvidia for the GPUs they produce [NVIDIA Corporation, nd]. This thesis uses CUDA and Nvidia hardware for GPU acceleration, and a basic introduction to CUDA programming is presented in this section.

CUDA has both C and C++ programming language support. In this thesis, we focus on the C extension, referred to as CUDA C, and access it through the PyCUDA Python wrapper [Klöckner, 2014].

4.1.1 Kernels, grids, blocks and threads

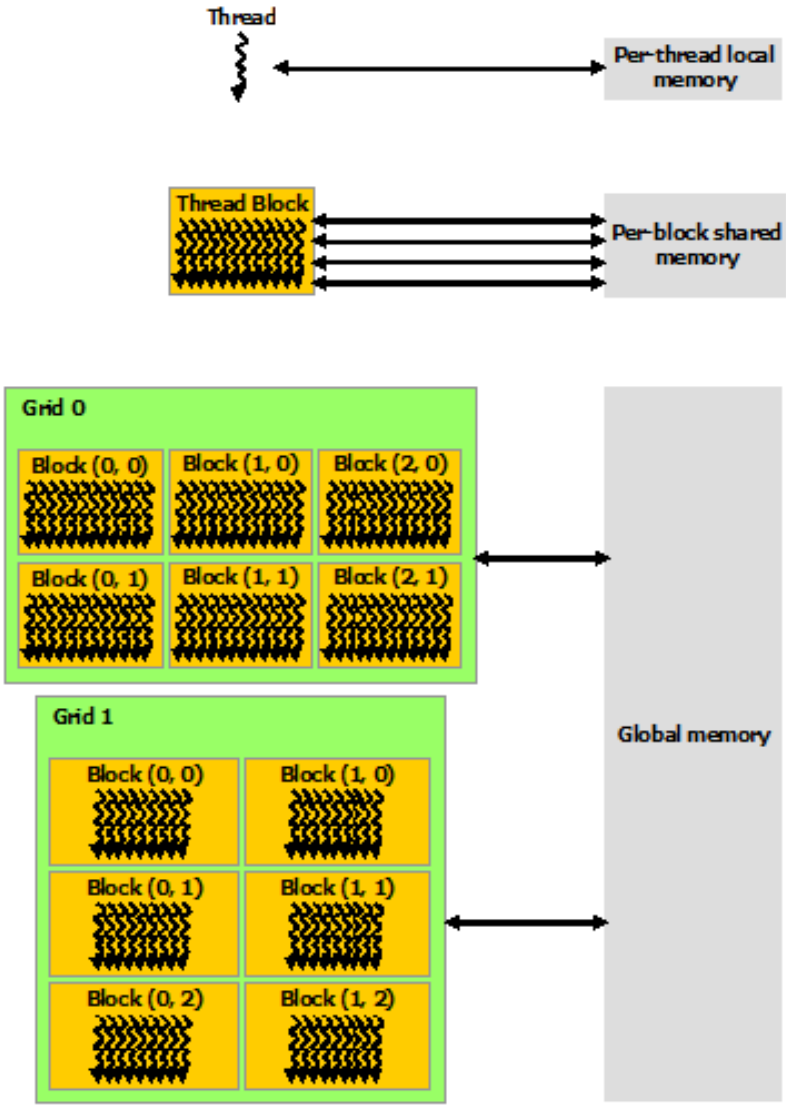
CUDA C introduces an extension to C through specialised functions called *kernels*, which are executed N times in parallel by N separate CUDA threads when called. In CUDA terminology, the GPU is referred to as the *device*, which is controlled by the *host*, typically the CPU on the system where the GPU is installed. Kernel execution and GPU threads are organised hierarchically. The kernels are executed on the device through a grid of *thread-blocks*, where each block contains a set of *threads*. The kernel is thus invoked by blocks of threads, where each thread runs the kernel once [NVIDIA Corporation, 2014a; Valdmanis, 2012]. Figure 4.1 shows an overview of this hierarchy along with the associated memory structure, which is described in Section 4.1.2.

In the CUDA C syntax, the kernels are defined as regular C functions with the addition of a `__global__` or `__device__` keyword. The former indicates that the function should be executable by a caller on either the host or device, while the latter is exclusively callable by the device. Each thread that executes a kernel is given a unique thread ID within its block, which may be accessed through the built-in `threadIdx` variable. If the kernel is called with a single thread block, `threadIdx` is sufficient to uniquely identify each thread. For multiple blocks, there are similar built-in variables used to access the block size and block ID: `blockDim` and `blockIdx`,

[1] <http://www.khronos.org/opencl/>

[2] Compute Unified Device Architecture

Figure 4.1: *Grids of thread-blocks with memory hierarchy [NVIDIA Corporation, 2014a]*



respectively. These block grids may be one-, two- or three-dimensional, as specified by the caller. For a kernel that accepts three-dimensional grids, a full thread address may be calculated as follows:

CUDA code

```
// Thread address for three-dimensional grids and blocks
int i = blockDim.x*blockIdx.x + threadIdx.x;
int j = blockDim.y*blockIdx.y + threadIdx.y;
int k = blockDim.z*blockIdx.z + threadIdx.z;
```

The kernel is called through an *execution configuration* syntax, `<<<Dg, Db, Ns, S>>>`, where `Dg` is the grid size (number of blocks), `Db` is the block size (threads per block), `Ns` is the number of bytes in dynamic shared block memory, and `S` is the associated CUDA stream[3]. `Ns` and `S` are optional arguments.

Basic sample code from the official CUDA documentation [NVIDIA Corporation, 2014a] shows how one might add the elements of two matrices A and B of size $N \times N$ and store the result in C:

CUDA code

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j]
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x,
                   N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

[3] CUDA streams provide a method for synchronising concurrent execution of multiple kernels.

In this code example, each of the 256 threads in each of the $N \times N/256$ blocks that execute `MatAdd()` performs one pair-wise addition.

4.1.2 Memory structure

In the CUDA programming model, the device memory is hierarchically structured and organised into three primary memory spaces as defined by the Nvidia CUDA documentation [NVIDIA Corporation, 2014a]. An overview of the structure is shown in Figure 4.1.

Global memory: All threads can access the persistent global device memory, which is typically up to several gigabytes in modern GPUs. The host can copy input data into this memory before executing a kernel to be used by the threads. Likewise, the kernel will typically update this memory with its results, which can be copied back into host memory. Global memory accesses are generally slow, and kernels that perform heavy computation on input data may see a significant performance increase by copying data into shared memory or thread-local registers for calculations before copying them back to global memory.

Per-block shared memory: Shared memory can be accessed by all threads within the same block. Since this memory is on-chip, shared memory accesses have a much higher bandwidth and lower latency than local or global memory.

Per-thread local memory: Each thread has a small pool of dedicated local memory. Like global memory, this memory is high latency and low bandwidth [NVIDIA Corporation, 2014a]. Variables are only placed into local memory under specific conditions, such as large arrays which would consume too much register space, and any variable when the kernel uses more registers than available (known as *register spilling*).

There are additional specialised memories available – such as texture and surface memory – that will not be covered here.

4.1.3 Thread execution

Nvidia GPUs have a number of multiprocessors, each of which executes in parallel with the others. With Nvidia's Kepler microarchitecture, each multiprocessor consists of 12 groups of 16 stream processors, where stream processors are commonly known as *cores*. The Nvidia GeForce GTX TITAN

has 14 such multiprocessors for a total of 2688 cores (see Section 4.2). Each core can execute a sequential thread, but the cores are executed in a method called *SIMT* (single-instruction, multiple-thread), where all cores in the same group execute the same instruction simultaneously [Cook, 2013, pp. 204—205].

The cores themselves are executed in *warps*, which are groups of 32 threads. To maximise performance, all threads in each warp should ideally execute the same instructions. However, the SIMT execution model allows for some flexibility at the expense of performance by serialising execution upon divergent instructions. An example of this flexibility is through *execution divergence* [Cook, 2013; Valdmanis, 2012].

Execution divergence occurs when diverging instruction paths are encountered. Typically these arise from altering the program control flow through conditional branching, for instance with if-else branches or loop conditions. Each path will be executed in turn until the control flow converges once more [Cook, 2013, p. 205].

4.1.4 The PyCUDA wrapper

PyCUDA is a complete Python wrapper to Nvidia's CUDA library [Klöckner, 2014]. Accessing CUDA functionality through Python allows for executing parallel code on Nvidia hardware through interactive scripting. PyCUDA also provides automatic memory clean-up, abstractions for compiling and running CUDA source code, and functionality for allocating and accessing memory on the device through NumPy array objects.

A simple example of usage is given by the official PyCUDA documentation [Klöckner, 2014]:

Python code

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy as np

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
```

```
""")

multiply_them = mod.get_function("multiply_them")

a = np.random.randn(400).astype(np.float32)
b = np.random.randn(400).astype(np.float32)

dest = np.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

In this code sample, the elements of two arrays of 400 pseudo-random single-precision floating point numbers are multiplied in parallel before printing the results.

The CUDA code is sent as a string to PyCUDA's `SourceModule` class, which invokes `nvcc`[4] to compile the source code into CUDA machine code. A PyCUDA reference to the CUDA kernel is retrieved from the compiled module and stored in `multiply_them` as a callable Python function reference.

The data used to communicate between the host and device is stored as NumPy array objects, which store their data in fixed-size blocks of memory in a C-contiguous memory layout [NumPy, 2014] compatible with CUDA.

4.2 Graphics card specifications

Simulations in later sections of this thesis have been run on one of two GPUs: a weaker consumer-grade Nvidia GeForce GT 650M for notebooks and a powerful Nvidia GeForce GTX TITAN. Some key specifications for these units are detailed in Table 4.1.

[4] Nvidia CUDA Compiler

[5] <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan>

[6] <http://www.geforce.com/hardware/notebook-gpus/geforce-gt-650m>

Table 4.1: *Key graphics card specifications*[5] [6] *[NVIDIA Corporation, 2012]*

Metric	GeForce GT 650M	GeForce GTX TITAN
GPU architecture	GK107	GK110
Process size	28 nm	28 nm
Transistors	1270 million	7080 million
Multiprocessors	2	14
CUDA cores	384 cores	2688 cores
Graphics clock	Up to 900 MHz	Up to 876 MHz
Effective memory clock	1800 MHz	Up to 6008 MHz
Texture fill rate	Up to 27.2 billion/s	187.5 billion/s
Peak performance	652.8 GFLOPS	4494 GFLOPS
Memory	1024 MB DDR3	6144 MB GDDR5
Memory interface width	128-bit DDR3/GDDR5	384-bit GDDR5
Memory bandwidth	Up to 80.0 GB/s	288.4 GB/s

5. Gotran

5.1 Overview

Gotran[7] provides a framework that aims to solve some of the issues that arise from working with complex ODE systems. It uses a domain-specific language to programmatically describe and declare an ODE system's right hand side function in a dedicated Gotran *form file* (Section 5.3). Instead of employing a specific ODE solver to solve the ODEs, Gotran allows for translating the form file to a number of different target languages for further integration into existing ODE solver softwares. See Figure 5.1.

As of this writing, Gotran has several features in current development, some of which were the main focus of this master thesis. In Figure 5.1 the items with solid borders represent currently implemented features.

Desired works are represented with dashed borders and include implementing support for more outputs: LaTeX, FORTRAN and CellML [Miller et al., 2010], Odeint [Ahnert and Mulansky, 2011], Sundials [Hindmarsh et al., 2005] and (Py)CUDA.

For this thesis, LaTeX output (Section 7) and GPU-acceleration via CUDA

[7] General *ODE Tran*slator

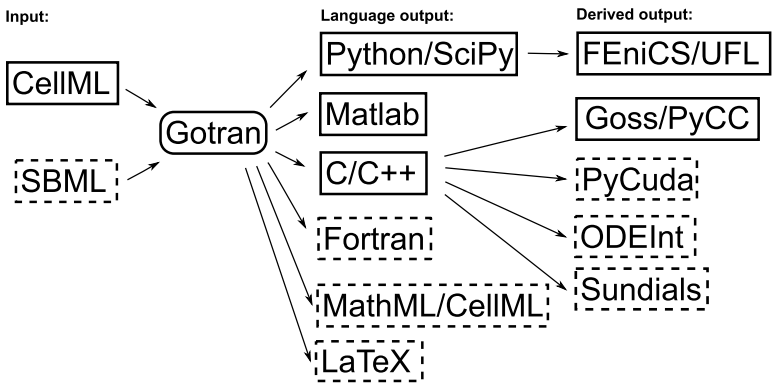


Figure 5.1: *Gotran as a hub for ODE handling*

(Section 8) have been implemented.

5.2 Strengths

Gotran is written in the Python programming language. There are several compelling reasons for this choice:

- Python's flexibility as a powerful and dynamic interpreted language simplifies the form files required to describe ODE systems. With Python and the Gotran framework, Gotran form files can be executed directly as Python code. This alleviates the need for new syntax and a separate parser to describe ODE systems. Additionally, the ODEs and their accompanying states, parameters and components can be loaded directly into memory as Python objects. These objects can then be used freely in further simulation, translation, modification and integration of the ODE systems in Python scripts.
- Python is one of the primary programming languages used in scientific computing [Langtangen, 2008], the field in which Gotran is intended to be used.
- Python has an extensive list of libraries conducive to scientific computing and to the specific needs of Gotran. These include NumPy/SciPy[8], ScientificPython[9], SymPy [SymPy, 2014] and PyCUDA (see Section 4.1.4) among others.

[8] <http://docs.scipy.org/doc/>

[9] <http://dirac.cnrs-orleans.fr/ScientificPython/ScientificPythonManual>

Gotran uses functionality from SymPy, a well-known and extensively tested Python library for handling symbolic mathematics, including:

- Automatic differentiation of expressions, which is used in automatic generation of the jacobian matrix of an ODE's partial derivatives.
- Extraction of common sub expressions, which is useful to generate efficient code.
- Support for code generation to C, Python, LaTeX and other outputs.

5.3 The Gotran form file

In Gotran, ODE systems are described in dedicated form files using Python syntax. In these files, the ODE's parameters and initial state values are defined, along with the derivative components utilising these values.

The states and parameters defined in the form file may use functionality from the modelparameters[10] Python package to attach metadata such as physical units and descriptions. The derivative components use the powerful SymPy library to describe and manipulate equations symbolically directly in Python.

Basic examples of state, parameter and derivative component definitions in a Gotran form file are shown below:

Python code

```
# Example of ODE state definitions in Gotran
states("Calcium dynamics",
       R_prime = ScalarParam(0.8978,
                             description="RyR availability"),
       Ca_i = ScalarParam(0.000153, unit="mM",
                          description="Intracellular Ca"),
       Ca_SR = ScalarParam(4.272, unit="mM",
                           description="SR Ca"),
       Ca_ss = ScalarParam(0.00042, unit="mM",
                           description="Subspace Ca"))

# Example of an ODE parameter definition in Gotran
parameters("Calcium pump current",
           g_pCa = ScalarParam(
              0.1238, unit="pA*pF**-1",
              description="I_pCa base conductivity"),
           K_pCa = ScalarParam(
              0.0005, unit="mM",
```

[10] <https://launchpad.net/modelparameters>

```
                description="Ca_i dissociation constant for
                    I_pCa"))

# Example of an ODE derivative component group in Gotran
component("Fast sodium current", "h gate")
h_inf = 1/(1 + exp((V + 71.55)/7.43))**2
alpha_h = Conditional(Lt(V, -40), 0.057*exp(-(V + 80)/6.8), 0)
    # ms**-1
beta_h = Conditional(Lt(V, -40), 2.7*exp(0.079*V) +
    310000*exp(0.3485*V), 0.77/(0.13*(1 + exp((V +
    10.66)/-11.1)))) # ms**-1
tau_h = 1/(alpha_h + beta_h) # ms
dh_dt = (h_inf - h)/tau_h
```

Note especially the last line, defining `dh_dt`. In Gotran form files, `dX_dt` is the dedicated syntax for declaring time derivatives of a state variable `X`.

See Section 14.1 for a full example of a complete form file. We will be using the Ten Tusscher-Panfilov model as defined by this form file for several other examples and demonstrations in Part II and Section 9.

5.4 Gotran `ODE` objects

Gotran stores ODE systems described by form files in specialised `ODE` objects. These objects store all initial state values, parameter values and symbolic expressions describing the ODE system, as well as additional metadata such as variable descriptions and component group labels.

Loading a form file into an `ODE` object is done through the `load_ode` function:

Python code

```
from gotran import load_ode
ode = load_ode('tentusscher_panfilov_2006_M_cell.ode')
```

As an example of the `ODE` object's contents, `ode.state_expressions` will, after loading a form file, contain a list of objects describing the derivative expressions for each state in the ODE model. A string representation of the derivative expression for one such state, the membrane potential "V", is as follows:

Python code

```
StateDerivative(State('V', ScalarParam(-85.423, unit='mV',
    description='Membrane potential'), Time('t',
    ScalarParam(0.0, unit='ms'))), -i_K1 - i_Ks - i_b_Ca -
    i_p_Ca - i_Kr - i_p_K - i_Stim - i_CaL - i_NaK - i_b_Na -
    i_NaCa - i_Na - i_to)
```

If we are interested in the definition of specific variables in the symbolic expression, they are accessible through the `ODE` object's `get_object` method. For instance, `ode.get_object('i_CaL')` returns an object describing the intermediate expression `i_CaL`:

Python code

```
(ODEComponent('L_type Ca current'),
 Intermediate('i_CaL',
     4*g_CaL*(F*F)*(0.25*Ca_ss*exp(2*F*V_eff/(R*T)) -
     Ca_o)*V_eff*d*f*f2*fCass/(R*T*(-1 +
     exp(2*F*V_eff/(R*T))))))
```

5.4.1 Solver components

Gotran also has functionality for generating and storing symbolic ODE solver algorithms through solver component objects. These solver components take an `ODE` object as input and return an encapsulated Python object containing the logic for computing one iteration of a numerical algorithm for an ODE system in the form of symbolic expressions.

5.5 GOSS

GOSS[11] is a separate Python and C++ library that provides a framework for solving ODE systems. It uses Gotran functionality for code generation and ODE representation through a Python interface, and provides tools to explicitly solve ODE systems.

[11] General ODE System Solver, a successor to PyCC [Mardal et al., 2007]

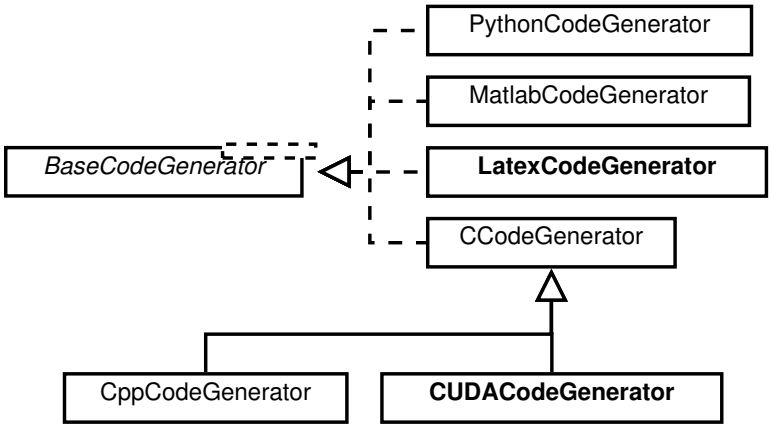


Figure 6.1: *Primary Gotran code generation classes – author's contributions in bold*

Part II
Contributions

6. Project implementation

This section briefly outlines the requirements for this thesis.

6.1 Automatically generated LaTeX output of ODE systems

Automatic generation of LaTeX output would eliminate a large source of error when presenting manuscripts with computations – especially typographical errors. By using the same Gotran form file to generate both the executable ODE simulation code and the corresponding LaTeX markup code describing the ODE system, this source of error may be minimised and possibly eliminated. Depending on the structure of the manuscript, the mathematical expressions may be presented in different formats.

We have implemented a number of different output formats along with an intuitive command-line interface to generate them. The implementation details are described in Section 7.

6.2 GPU acceleration

GPU acceleration interfaced via Python is useful for research dependent on high-performance computing. When large-scale ODE systems are uncoupled, they can be solved simultaneously in parallel, as the computation of such systems is a *parallel problem* (see Section 8.1). However, maximizing performance from the generated GPU code is a non-trivial problem.

"The development of additional cell models for the GPU requires significant technical skill in comparison with coding the same model for the CPU. [...] it would thus be desirable to develop a tool for automatically generating GPU code and add this functionality to existent cell model repositories." [Vigueras et al., 2014, p. 131].

This thesis implements GPU acceleration using CUDA, with support for multiple solver algorithms through Gotran and GOSS. The implementation is detailed in Section 8, and test results are documented in Section 9.

7. LaTeX output from Gotran

A common way to publish a developed ODE model is to describe it in a manuscript to be included in the publication. However, translating a complex model from a source file to a LaTeX markup file is tedious and error prone. One solution to this problem is to let Gotran not only generate executable code for solving the ODEs, but also automatically generate well-formatted LaTeX markup code ready to be included into a LaTeX manuscript either verbatim, or with minimal changes.

For this thesis, we have developed such a tool integrated with Gotran. This section describes its implementation and usage. A full generated document of the Ten Tusscher-Panfilov model can be found as Figure 14.1 in Section 14.2 in the Appendix.

7.1 Implementation overview and overall structure

The generation of the LaTeX code is contained within the `LatexCodeGenerator` Python class in Gotran. The overall structural design strategy was to create a hierarchical set of string templates into which formatted expressions, descriptions, variable names, package imports, formatting options and even subtemplates are injected, starting with the root template:

Python code

```
_latex_template =
    """\\documentclass[a4paper,{FONTSIZE}pt]{{article}}
{PKGS}
{PREOPTS}

\\begin{{document}}
{OPTS}
{BODY}
{ENDOPTS}
\\end{{document}}"""
```

After initialising an instance of the class with a Gotran `ODE` object and optional generation parameters (Section 8.9), it is ready to generate a complete document from the root template, or individual components from similarly structured subtemplates. For a full document, the generated LaTeX code consists of three major sections:

- A table of parameter values
- A table of initial state values
- The derivative components containing the equations describing the ODE system

7.2 State and parameter table generation

Table 7.1: *Abbreviated example of a generated parameter table*

Parameter	Value	Description
K_{pCa}	500×10^{-6} mM	Ca_i dissociation constant for I_{pCa}
g_{pCa}	0.12 pA pF^{-1}	I_{pCa} base conductivity

Table 7.2: *Abbreviated example of a generated state table*

State	Value	Description
Ca_i	153×10^{-6} mM	Intracellular Ca
R_{prime}	0.90	RyR availability
Ca_{SR}	4.27 mM	SR Ca
Ca_{ss}	420×10^{-6} mM	Subspace Ca

The state table describes the initial values of the variables in the ODE

system, while the parameter table describes values which are constant with respect to later ODE calculations.

7.2.1 Table implementation considerations

Using the basic LaTeX `table` construct became problematic for ODE systems with too many states and parameters to easily fit on one page. The `table` would bleed off the bottom of the page without transitioning to the next. A few options were explored to solve this issue.

The first, naïve solution was to dump the state and parameter values into a standard LaTeX `table` construct, and leave the user to manually split the data to fit on each page. This solution had the benefit of implementation simplicity and robustness. As we aimed to maximise automation, however, this option was not ideal for large tables.

As our second option, we could still have employed the standard LaTeX `table` constructs, but additionally relied on program logic during code generation to automatically split the data into separate tables upon detecting that it is too large to fit the page. There are, however, several implementation and usability issues that arise from such an approach. For the implementation, this approach would have been unnecessarily complex to develop. Determining the maximum amount of data that fits on one page in a LaTeX table would rely on far too many variables to easily be implemented in Python in a robust manner. We would be forced to account for page margins, font type, font size, potentially tall rows for arbitrarily long variable descriptions that wrap multiple lines, and other factors. Even if such a solution were implemented, the result would have been tables of fixed length hard-coded into the final LaTeX code. This would still not have been satisfactory from a usability standpoint, as users would be forced to manually maintain and change these tables if they altered font sizes, font families, variable descriptions, the table's position within the document and more.

To combine the benefits of implementation simplicity and the ease-of-use and flexibility from automatic table splitting, our third and final solution uses a more feature-rich LaTeX table construct than the standard `table`. We found that the `longtabu` table construct, provided by the `longtable` [Carlisle, 2004] and `tabu` [Chervet, 2011] LaTeX packages, best suited our needs. This table construct gave us several advantages:

Simplicity: Once we have formatted the data, we can trivially dump it into the `longtabu` table.

Flexibility: From a usability standpoint, the user may then reposition the table, refactor the data and perform a variety of changes. The table will dynamically adjust to these changes automatically.

7.2.2 Implementation details

The tables have been implemented by creating Python string templates to supply the boilerplate code. The template is then populated with data by the `LatexCodeGenerator` class. The parent template for the parameter table is as follows:

Python code

```
_param_table_template = """
% ---------------- BEGIN PARAMETERS ---------------- %

\\{SECTIONTYPE}*{{Parameters}}\n
\\label{{sec:ODE_Parameters}}
{OPTS}
\\begin{{longtabu}}{{| l l {PDESCCELLSTYLE} |}}
  \\caption[Parameter Table]{{%
    \\textbf{{Parameter Table}}}}\\\\
  \\hline
  \\multicolumn{{1}}{{|c}}{{%
    \\textbf{{Parameter\\hspace{{0.5cm}}}}}} &
  \\multicolumn{{1}}{{c}}{{%
    \\textbf{{Value\\hspace{{0.5cm}}}}}} &
  \\multicolumn{{1}}{{c|}}{{%
    \\textbf{{Description\\hspace{{0.5cm}}}}}}\\\\ \\hline
  \\endfirsthead
  \\multicolumn{{3}}{{c}}%
    {{{{\\bfseries\\tablename\\%
      \\thetable{{}} --- continued from previous page}}}}
  \\\\ \\hline
  \\multicolumn{{1}}{{|c}}{{%
    \\textbf{{Parameter\\hspace{{0.5cm}}}}}} &
  \\multicolumn{{1}}{{c}}{{%
    \\textbf{{Value\\hspace{{0.5cm}}}}}} &
  \\multicolumn{{1}}{{c|}}{{%
    \\textbf{{Description\\hspace{{0.5cm}}}}}}\\\\ \\hline
  \\endhead
  \\hline
```

```
  \\multicolumn{{3}}{{|r|}}%
    {{{{Continued on next page}}}}\\\\ \\hline
  \\endfoot
  \\hline \\hline
  \\endlastfoot
  {BODY}\\\\
  \\hline
\\end{{longtabu}}
{ENDOPTS}

% ----------------- END PARAMETERS ----------------- %
"""
```

The template for generation of the initial state values table is equivalent.

The parameter table template may then be populated with a few simple functions:

Python code

```
def generate_parameter_table(self, params=None):
    """
    Return a LaTeX-formatted string for a longtable
    describing the ODE's parameters.
    """
    params = params if params else self.params
    param_str = "\\\\\\n".join(
        self.format_param_table_row(par)
        for par in self.ode.parameters)
    param_table_opts =
       self.format_options(exclude=["page_columns"])
    param_table_output = _param_table_template.format(
        SECTIONTYPE=params["section_type"],
        PDESCCELLSTYLE=params[
            "parameter_description_cell_style"],
        OPTS=param_table_opts["begin"], BODY=param_str,
        ENDOPTS=param_table_opts["end"])
    return param_table_output

def format_param_table_row(self, param):
    """
    Return a LaTeX-formatted string for a longtable row
    describing a parameter.
    E.g.:
    >>> LatexCodeGenerator.format_param_table_row(
    ...     Parameter("g_earth", ScalarParam(9.81,
    ...         unit="m/s**2", description="Surface gravity"))
```

```
    '  $g_{earth}$\\hspace{0.5cm} & $9.81
      \\mathrm{\\frac{m}{s^{2}}}$
      \\hspace{0.5cm} & Surface gravity'
    """
    return "  ${NAME}$\\hspace{{0.5cm}} & {VAL}" \
           "\\hspace{{0.5cm}} & {DESC}".format(
                        NAME=self.format_expr(param.name),
                        VAL=param._repr_latex_(),
                        DESC=self.format_description(
                                param.param.description,
                                param.name))
```

We used the SymPy and modelparameters Python packages to convert state and parameter names and values from an internal representation to a valid LaTeX string. Variable names are treated as single atomic units of a mathematical expression, and we run them through a simple `format_expr` function to prepare them for conversion through the aforementioned packages:

Python code

```
def format_expr(self, expr):
    """
    Return a LaTeX-formatted string for a sympy expression.
    E.g.:
    >>> LatexCodeGenerator.format_expr("exp(i*pi) + 1")
    'e^{i \\pi} + 1'
    """
    if isinstance(expr, str) and expr in _greek:
        return "\\{0}".format(expr)
    # Some values are treated as special cases by
       sympy.sympify.
    # Return these as they are.
    if isinstance(expr, str) and expr in \
            filter(lambda x: len(x) == 1, dir(sympy)):
        return expr
    return modelparameters.latex(
        sympy.sympify(expr), **self.print_settings)
```

7.3 Component generation

The components describe the right hand side of the ODE system by means of other states and parameters. These components are organised into separate groups as dictated by the given Gotran form file (see Section 5.3),

Fast sodium current

$$h_{inf} = \frac{1}{\left(1 + 15.2 \times 10^3 e^{0.14V}\right)^2} \tag{7.1a}$$

$$\alpha_h = \begin{cases} 443 \times 10^{-9} e^{-0.15V} & \text{for } V < -40 \\ 0 & \text{otherwise} \end{cases} \tag{7.1b}$$

$$\beta_h = \begin{cases} 2.7e^{79\times10^{-3}V} + 310 \times 10^3 e^{0.35V} & \text{for } V < -40 \\ \frac{0.77}{0.13+49.8\times10^{-3}e^{-90.1\times10^{-3}V}} & \text{otherwise} \end{cases} \tag{7.1c}$$

$$\tau_h = \frac{1}{\beta_h + \alpha_h} \tag{7.1d}$$

$$\frac{dh}{dt} = \frac{1}{\tau_h}\left(-h + h_{inf}\right) \tag{7.1e}$$

Figure 7.1: *Example of a generated component group*

$$\frac{dV}{dt} = -i_{K1} - i_{Ks} - i_{bCa} - i_{pCa} - i_{Kr} - i_{pK} - i_{Stim} - i_{CaL} - i_{NaK} - i_{bNa} - i_{NaCa} - i_{Na} - i_{to} \tag{7.2}$$

(a) *Using* `equation`

$$\frac{dV}{dt} = -i_{K1} - i_{Ks} - i_{bCa} - i_{pCa} - i_{Kr} - i_{pK} - i_{Stim} - i_{CaL} - i_{NaK} - i_{bNa} - i_{NaCa} - i_{Na} - i_{to} \tag{7.3}$$

(b) *Using* `dmath`

Figure 7.2: *Demonstration of automatic linebreaks with* `dmath` *vs.* `equation`

where the equations themselves are also defined. A well-structured ODE system definition will include numerous intermediate calculations for appropriate component groups based on the complexity of the equation for a state's derivative function.

Figure 7.1 shows a generated sample component group from the Ten Tusscher-Panfilov model with equations organised into two columns.

7.3.1 Automatic linebreaks

Similar to our reasoning about pagebreaks for tables, it is necessary to make some design choices towards maximising automation when ensuring readability in the final document. Even if we assume well-formed equations from the Gotran form file with reduced complexity, some equations may not be possible to cleanly display on a single line regardless. Not accounting for this would pose a major readibility issue.

Using common LaTeX constructs such as `equation` and `align` would prove to be problematic for reasons similar to the `table` construct in Section 7.2.1. They would require deliberately inserting linebreaks at appropriate breakpoints in long equations to prevent them from bleeding off the page, either manually by the user, or programmatically with program logic. As before, we found the best solution was to use an appropriate LaTeX package such as `breqn` [Høgholm et al., 2012], which "facilitates automatic line-breaking of displayed math equations". See Figure 7.2 for a comparison between using `equation` and `breqn`'s `dmath` for the following generated LaTeX code:

LaTeX code

```
\frac{dV}{dt} = - i_{K1} - i_{Ks} - i_{b Ca} - i_{p Ca} -
    i_{Kr} - i_{p K} - i_{Stim} - i_{CaL} - i_{NaK} - i_{b Na}
    - i_{NaCa} - i_{Na} - i_{to}
```

Depending on the specific structure of the equation and the chosen constraints on page width, the `dmath` construct may fail to properly break up an equation. For our use cases, this appears to occur most often with convoluted conditional equations. In such cases, the user is forced to manually adjust or refactor the expression to their preference. An example of this may be seen in Section 14.2. For a Gotran form file with well-structured equations, this occurs infrequently enough that we will consider the failure rate to be within acceptable levels.

7.3.2 Implementation details

The generation of components also uses string templates to simplify the implementation, and is primarily contained within one main function `generate_components`:

Python code

```
def generate_components(self, params=None):
    """
    Return a LaTeX-formatted string of the ODE's derivative
        components and
    intermediate calculations.
    """
    params = params if params else self.params
    components_str = ""
    comp_template = "{LABEL}\n\\label{{comp:{LABELID}}}\n" \
        "\\begin{{dgroup{SUBNUM}}}\n" \
        "{BODY}\\end{{dgroup{SUBNUM}}}\n"
    eqn_template = \
        "  \\begin{{dmath}}\n    \\label{{eq:{0}}}\n" \
        "    {1} = {2}\\\\\n  \\end{{dmath}}\n"

    subnumbering = '' if params["equation_subnumbering"] \
                      else '*'

    for comp in self.ode.components:
        body = [obj for obj in comp.ode_objects
                if isinstance(obj, Expression)]

        if not body:
            continue

        format_label = self.format_component_label(comp.name)
        label_id = comp.name.replace(' ', '_')
        format_body = ""

        # Iterate over all objects of the component
        for obj in body:
            format_body += eqn_template.format(
                obj.name,
                obj._repr_latex_name(),
                obj._repr_latex_expr())

        components_str += \
            comp_template.format(LABEL=format_label,
```

```
                                    LABELID=label_id,
                                    BODY=format_body,
                                    SUBNUM=subnumbering)

    components_opts = \
        self.format_options(override=["page_columns",
            "math_font_size"])
    components_output = _components_template.format(
        SECTIONTYPE=params["section_type"],
        OPTS=components_opts["begin"],
        BODY=components_str,
        ENDOPTS=components_opts["end"])
    return components_output

def format_component_label(self, label):
    """
    Return a LaTeX-formatted string of an ODE component group
        label.
    """
    label_opts = self.format_options(
        override=["bold_equation_labels"])
    return "{0}{1}{2}\\\\".format(
        label_opts["begin"],
        label.replace("_", "\\_"),
        label_opts["end"])
```

`generate_components` iterates over each component in the Gotran `ODE` object. For each component, the associated Gotran `Expression` objects are collected. The `Expressions` represent the relevant derivative equations for the states and their related intermediate equations. These are subsequently formatted into valid mathematical LaTeX expressions through `modelparameters` and `sympy` functions before being inserted into appropriate subtemplates and configured with additional code generation parameters.

The example shown in Figure 7.1 was created with the following generated LaTeX code:

LaTeX code

```
\textbf{Fast sodium current}\\
\label{comp:h_gate}
{\fontsize{10.0}{12.0}
\begin{multicols}{2}
\begin{dgroup}
```

```
    \begin{dmath}
      \label{eq:h_inf}
      h_{inf} = \frac{1}{\left(1 + 15.2\!\times\!10 ^{3}
          e^{0.14 V}\right)^{2}}\\
    \end{dmath}
    \begin{dmath}
      \label{eq:alpha_h}
      \alpha_{h} = \begin{cases} 443\!\times\!10 ^{-9} e^{-
          0.15 V} & \text{for}\: V < -40 \\0 &
          \text{otherwise} \end{cases}\\
    \end{dmath}
    \begin{dmath}
      \label{eq:beta_h}
      \beta_{h} = \begin{cases} 2.7 e^{79\!\times\!10 ^{-3} V}
          + 310\!\times\!10 ^{3} e^{0.35 V} & \text{for}\: V <
          -40 \\\frac{0.77}{0.13 + 49.8\!\times\!10 ^{-3} e^{-
          90.1\!\times\!10 ^{-3} V}} & \text{otherwise}
          \end{cases}\\
    \end{dmath}
    \begin{dmath}
      \label{eq:tau_h}
      \tau_{h} = \frac{1}{\beta_{h} + \alpha_{h}}\\
    \end{dmath}
    \begin{dmath}
      \label{eq:dh_dt}
      \frac{dh}{dt} = \frac{1}{\tau_{h}} \left(- h +
          h_{inf}\right)\\
    \end{dmath}
  \end{dgroup}
  \end{multicols}
  }% end fontsize
```

7.4 Formatting and generation parameters

The implemented LaTeX code generator supports a number of user-specified generation parameters that control the formatting of the final document or the structure of the code. This includes font size for text and equations, page orientation and margins, multi-column equations and the omittance of a LaTeX preamble.

7.5 `gotran2latex`

`gotran2latex` is a simple script that provides a command-line interface to the implemented LaTeX code generator. It takes a Gotran form file as input, along with optional code generation options, and generates a `.tex` file with the resulting LaTeX code. Instructions for usage are available in Section 13.1.

8. GPU acceleration of ODE solvers

Gotran does not provide any means to explicitly solve ODEs. Instead it provides functionality to generate code which can interface with existing solver frameworks. Before the contributions from this thesis, Gotran provided support for outputs to Matlab and pure Python. It was also capable of generating code for a C/C++ function body, which can be placed into a function specific to a particular C/C++ solver library. Support for the in-house GOSS C++ library was already in place.

This thesis implements code generation to CUDA, Nvidia's language for parallel GPU programming, as an extension to Gotran. It also implements supplementary functionality in GOSS to use the resulting code interactively through PyCUDA.

8.1 Parallelisation on GPU

A common application for Gotran generated files is simulation of the electrophysiology of the heart. In such simulations, hundreds of thousands or millions of almost identical ODE systems need to be solved on each time step. Since these ODE systems do not share state values at any point, they are independent of each other. As such, this problem may be considered *embarrassingly parallelisable*. [Ackermann et al., 2009, p. 2]

The GOSS C++ library already uses OpenMP[12] to parallelise the solving process and provides significant speed-ups. Through OpenMP, GOSS utilises modern CPUs' multi-core architecture to achieve its speed-ups. We posit, however, that there are even more significant speedups to be gained by utilising the far more parallel structure of GPUs for solving ODEs. To achieve this, we tailored our code specifically for GPUs through an appropriate framework. In this thesis, we targeted Nvidia GPUs and the accompanying CUDA framework.

[12] `<http://openmp.org/wp/>`

8.2 CUDA generation and a PyCUDA solver interface

Since all our existing tools can be interfaced using Python, it was natural to interface our implementation with PyCUDA [Klöckner, 2014]. PyCUDA is a complete Python wrapper to Nvidia's CUDA library. Section 4.1.4 provides a brief description of the PyCUDA library.

We have focused on efficiency and explored various methods for optimisation of the algorithms for these solvers, specifically tailored for GPU architecture. Automatic generation of CUDA code for solvers of given ODE models has been implemented Gotran. The existing C code generator in Gotran served as a good basis for the development of the CUDA code generation, since CUDA itself is accessible through libraries and extensions to C/C++.

An interface to use the CUDA code generator to explicitly solve ODEs on GPU has also been implemented as an extension to the GOSS framework. The generated solver code is compiled and loaded onto the GPU, and subsequently accessed using the PyCUDA wrapper API in Python through our GOSS interface. An example of usage can be seen in Section 8.11.

8.3 Implementation overview

The implementation is split into two primary components:

CUDA code generation: Implemented as a part of Gotran, this generates generic CUDA code for calculating the right-hand-side function of an ODE model given a specific ODE solver method.

PyCUDA ODE solver: Implemented as a part of GOSS, this invokes the `CUDACodeGenerator` to generate the code for Nvidia's CUDA platform. By wrapping the generated code with PyCUDA, it allows a script, library or interactive user to compile the CUDA code, load the binaries onto an Nvidia graphics card and run the simulation within a Python session.

As mentioned briefly in Section 8.1, the simulated ODE system consists of a large number of almost identical ODEs. Specifically, this refers to their parameters generally being identical and their states generally undergoing nearly identical calculations. There are some notable exceptions outlined in Section 8.4.

8.4 Field states and field parameters

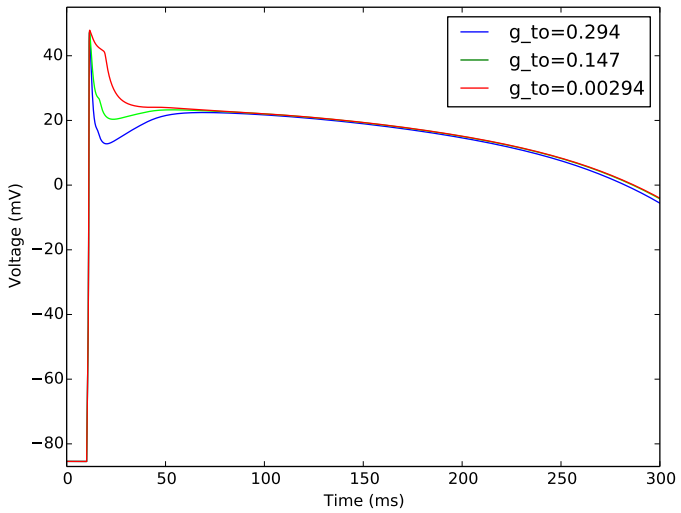


Figure 8.1: *Effect of transient outward current g_{to} on the Ten Tusscher-Panfilov transmembrane potential*

Local memory and local registers on GPU threads are limited, and copying memory between host (CPU) and device (GPU) is time-consuming. In an effort to conserve memory usage, we will reserve states and parameters to the GPU by default and not directly expose them to the Python code running the simulation on the host. The user may, however, find it useful to select specific states and parameters of interest for further calculations on the host, referred to as *field states* and *field parameters* respectively. This has a few important implications and side effects.

8.4.1 Field states

It is often useful to track one or several states of interest for further analysis and calculations of their values as they change over time. These states are marked as field states. In our use cases, this is most commonly for tracking and plotting the transmembrane voltage during an action potential event. A more complete simulation of a cardiac model will also require performing calculations to solve PDEs on select field states for each time step of the simulation.

These are CPU-bound operations performed on the host. It is therefore

necessary to transfer these field states between GPU memory into CPU memory on each time step. For each iteration, the host will therefore perform the following steps:

- Transfer current field state values from the host (CPU) to the device (GPU).
- Execute the CUDA kernel to compute one time step of the ODE system on GPU.
- Transfer updated field state values from the device to the host.
- Perform additional calculations of the field states on the host (e.g. solve PDEs or store field states for later plotting/analysis).
- Increase the current simulation time by the time step.

8.4.2 Field parameters

Regular ODE parameters are defined in the Gotran form file and remain constant throughout the duration of the simulation, and are identical for all simulated nodes (ODEs) in the ODE system. For this reason, each parameter is not stored separately for each node, but as a single, constant value shared by all nodes. It is useful, however, to set up certain parameters with values that vary across the ODE system. These parameters are marked as field parameters. The field parameters may represent variations in properties of separate cells, where each cell is represented by one node in the simulation.

While parameters are initialised once for all nodes as the ODE model is loaded from a Gotran form file, each field parameter may be initialised by the user with values that differ between nodes. The field parameters may then remain static throughout the simulation, or get updated between iterations in a similar manner to field states.

Figure 8.1 shows the transmembrane potential of the Ten Tusscher-Panfilov model for different values of the g_{to} parameter, representing the base conductance of transient outward current. Its original value as defined in the model's Gotran form file was 0.294 nS/pF.

8.4.3 Effects on memory usage and memory transfer

Since all states must be used in calculations separately for each node in the ODE system, each state will already be stored separately for each node. Marking a state as a field state will therefore not increase memory

usage on the GPU. In fact, field states do not introduce any changes to the CUDA kernel that computes the ODE system's right-hand-side (the forward function). For the host to interact with the field states in a meaningful manner, however, an array to hold the field state data must be allocated on the CPU large enough to hold one floating point number for each field state on each node. Additionally, as field states are copied between the host and device on each time step, we expect there to be a minor performance hit due to the increased data transfer between the CPU and GPU.

Marking an ODE parameter as a field parameter, however, does increase memory usage on the GPU. Whereas regular, static parameters only need to be stored once across all nodes, each field parameter is stored separately for each node. If the host code updates the field parameters between time steps as well, field parameters will see an increase in data transfer similar to that of field states.

8.5 Implementation of CUDA code generation

A `CUDACodeGenerator` class has been implemented for this thesis. It generates CUDA code for initiating and solving an ODE. The primary components of this generated code are:

- initialisation of states, field states and field parameters
- kernel functions for external retrieval and updating of field states
- a forward kernel function that computes one time step of the ODE's right-hand-side (see Section 3)

`CUDACodeGenerator` has been implemented as a subclass of Gotran's `CCodeGenerator`, which itself is a subclass of `BaseCodeGenerator`. As CUDA is an extension of C/C++, it is appropriate for the `CUDACodeGenerator` to reuse relevant functionality from the parent class which already generates output for ODE solvers in C. There are, however, some crucial differences the CUDA code generator needs to handle.

First, we must be able to generate a single CUDA forward function to compute the values of potentially millions of almost identical ODEs on hundreds of separate simultaneous threads using the same code. Optimally, this should also be achieved with minimal conditional branching of program flow to prevent execution divergence due to the GPU's SIMT architecture (see Section 4.1.3).

Additionally, we have implemented functions for handling initialisation of field states and field parameters, and for cleanly transferring field states between the host CPU and the GPU device.

8.5.1 Forward function

The forward function computes one time step of the ODE's using an ODE solving algorithm such as those presented in Section 3. The main generation of the forward function is handled by the `function_code` method in `CUDACodeGenerator`:

Python code

```
def function_code(self, comp, indent=0,
                  default_arguments=None,
                  include_signature=True,
                  return_body_lines=False):
```

The primary argument is `comp`. This is a Gotran solver component, such as `RushLarsen`, which generates a collection of Gotran `Expressions` and supplementary data to compute one step of a specific ODE solver algorithm. These `Expressions` make heavy use of SymPy to allow for a symbolic representation of mathematical expressions within Python.

The arguments supplied by the called are first read and validated. The code for this has been omitted. After validation, the initial lines of the body of the forward function are set:

Python code

```
# Initialization
body_lines = [
    "const int thread_ind = blockIdx.x*blockDim.x + "
    "threadIdx.x"]
body_lines.append(
    "if (thread_ind >= n_nodes) return; "
    "// number of nodes exceeded")
body_lines.append(
    "const int {0}_offset = thread_ind*{1}".format(
        states_name, comp.root.num_full_states))

if len(field_parameters) > 0:
    body_lines.append(
        "const int {0}_offset = thread_ind*{1}".format(
            field_parameter_name, len(field_parameters)))
```

```
body_lines.extend(self._init_arguments(comp))
```

Lines of code for the function body are collected in `body_lines`. These are later formatted and combined along with the function definition through supplementary code generation functions. Most code generation functions in Gotran use this method.

Since we are applying the same function to potentially millions of nodes, we need to differentiate between them based on which GPU block and thread is currently running the function. This is done in CUDA through the global `structs` `blockIdx`, `blockDim` and `threadIdx`. We use these values to calculate the current thread index, and store it in `thread_ind` in the resulting CUDA code. We also initialise useful offsets to determine where the current thread's data is stored in the global state and field parameter arrays.

Note that we organise our thread-blocks as one-dimensional, and we only use the x-coordinate of the thread and block indices. While CUDA supports up to three-dimensional grids, ODE systems are not inherently structured as such.

A version of `CCodeGenerator`'s `_init_arguments` modified to work with CUDA generates the code initialising each state, parameter and field parameter for the forward function. With default code generation parameters, this creates local copies of the values in the current thread's local registers. This is useful for optimisation purposes during computation, as retrieving these values from the thread registers requires far fewer clock cycles than fetching them from the single, large array in the global GPU memory. However, other generation options are available, as detailed in Section 8.9.

After initialisation, we generate the code for the groups of expressions which compute one step for each state through their derivatives. This is done by iterating over the expressions composed by the Gotran solver component and formatting them for our purposes:

Python code

```
# If named body representation we need to check for duplicates
duplicates = set()
declared_duplicates = set()
if params.body.representation == "named":
    collected_names = set()
    for expr in comp.body_expressions:
        if isinstance(expr, Expression) and \
```

```
                not isinstance(expr, IndexedExpression):
            if expr.name in collected_names:
                duplicates.add(expr.name)
            else:
                collected_names.add(expr.name)

# Iterate over any body needed to define the dy
for expr in comp.body_expressions:
    if isinstance(expr, Comment):
        body_lines.append("")
        body_lines.append("// " + str(expr))
        continue
    elif isinstance(expr, IndexedExpression):
        name = "{0}".format(self.obj_name(expr))
    elif expr.name in duplicates:
        if expr.name not in declared_duplicates:
            name = "{0} {1}".format(self.float_type,
                                    self.obj_name(expr))
            declared_duplicates.add(expr.name)
        else:
            name = "{0}".format(self.obj_name(expr))
    else:
        name = "const {0} {1}".format(self.float_type,
                                      self.obj_name(expr))
    body_lines.append(self.to_code(expr.expr, name))
```

To use named intermediate values, we must first declare and initialise them with a type. To do this we must keep a track of duplicate intermediate variables. These represent variables which are recalculated at some point after initialisation, and we must thus ensure that they are not subsequently redeclared after the initial declaration. If a variable instead is calculated only once, it is stored as a `const`.

Finally, we wrap the generated body in a function prototype, indent and split the lines and return the generated code as a string. The code for this is omitted.

As an example of the generated CUDA code, the following is a heavily abbreviated code snippet from generated code of the forward function for the Ten Tusscher-Panfilov ODE model using the Rush-Larsen algorithm, generated with single-precision floating point values:

CUDA code

```
// Compute a forward step using the rush larsen algorithm on
// the tentusscher_panfilov_2006_M_cell ODE
__global__ void forward_rush_larsen(float* d_states, const
    float t, const float dt, const float* d_parameters, const
    float* d_field_parameters, const unsigned int n_nodes)
{
  const int thread_ind = blockIdx.x*blockDim.x + threadIdx.x;
  if (thread_ind >= n_nodes) return; // number of nodes
      exceeded;
  const int d_states_offset = thread_ind*19;
  const int d_field_parameters_offset = thread_ind*2;

  // Assign states
  const float Xr1 = d_states[d_states_offset + 0];
  const float Xr2 = d_states[d_states_offset + 1];
  const float Xs = d_states[d_states_offset + 2];
  /* ... */
  const float K_i = d_states[d_states_offset + 18];

  // Assign parameters
  const float P_kna = d_parameters[0];
  const float g_K1 = d_parameters[1];
  const float g_Kr = d_parameters[2];
  /* ... */
  const float K_o = d_parameters[52];
  const float g_CaL =
      d_field_parameters[d_field_parameters_offset + 0];
  const float g_to =
      d_field_parameters[d_field_parameters_offset + 1];

  /* Expressions for 6 components omitted... */

  // Expressions for the Xs gate component
  const float xs_inf = 1.0/(1.f + exp(-5.f/14.f - V/14.f));
  const float alpha_xs = 1400.f/sqrt(1.f + exp(5.f/6.f -
      V/6.f));
  const float beta_xs = 1.0/(1.f + exp(-7.f/3.f + V/15.f));
  const float tau_xs = 80.f + alpha_xs*beta_xs;
  const float dXs_dt = (-Xs + xs_inf)/tau_xs;
  const float dXs_dt_linearized = -1.f/tau_xs;
  d_states[d_states_offset+2] = Xs + (fabs(dXs_dt_linearized) >
       1.0e-8f ?
    (-1.0f +
        exp(dt*dXs_dt_linearized))*dXs_dt/dXs_dt_linearized :
```

```
        dt*dXs_dt);

  /* Expressions for 22 components omitted... */
}
```

8.5.2 State and field parameter initialisation functions

The global array of states shared by all nodes is initialised in a separate function on the GPU. The CUDA code is generated by the following method:

Python code

```
def init_states_code(self, ode, indent=0):
```

It generates code that computes the current thread index and the array offsets in a manner similar to `function_code`, before setting up the main state array initialisation code:

Python code

```
  # Main body
  body_lines.extend(
      "{0}[{0}_offset+{1}] = {2}{3}; // {4}".format(
          array_name, i, state.init, float_str, state.name)
      for i, state in enumerate(ode.full_states))
```

The initial state values are retrieved from the given Gotran `ODE` object and is hardcoded into the generated code to populate the CUDA array. Each node's states are stored successively in the array, such that the ith node uses the data at indices $\{i \cdot N_s,\ i \cdot N_s + 1,\ \ldots,\ (i+1) \cdot N_s - 1\}$, where N_s is the number of states per ODE.

An abbreviated example of the resulting CUDA code for the Ten Tusscher-Panfilov model is as follows:

CUDA code

```
// Init state values
__global__ void init_state_values(double *d_states)
{
  const int thread_ind = blockIdx.x*blockDim.x + threadIdx.x;
  const int d_states_offset = thread_ind*19;
  d_states[d_states_offset+0] = 0.0165; // Xr1;
  d_states[d_states_offset+1] = 0.473; // Xr2;
  /* ... */
  d_states[d_states_offset+18] = 138.52; // K_i;
}
```

Note that double-precision floating points are used in this example. Single-precision float numerals are written with a trailing `f`, e.g. `0.0165f` as opposed to double-precision `0.0165`.

The initialisation of field parameters is done in a similar manner.

8.5.3 Field state getter and setter functions

The CUDA function for retrieving field states to GPU is generated by the `field_states_getter_code` function:

Python code

```
def field_states_getter_code(self, ode, indent=0):
```

As previously, the thread index and offsets are calculated before generating the main body of the function:

Python code

```
# Main body
body_lines.extend(
    "{0}[field_{2}_offset + {3}] = "\
    "{1}[{2}_offset + {4}]; //{5}".format(\
        field_array_name, array_name,
        base_array_name, i, states.index(state), state.name)
    for i, state in enumerate(field_states))
```

Here, code is generated so the field states array stored on CPU can be updated with the values from the current GPU states array. The host array is referred to by `field_array_name`, while the GPU device array is denoted by `array_name`.

The resulting CUDA code for the Ten Tusscher-Panfilov model with one field state (the transmembrane potential "V") is as follows:

CUDA code

```
// Get field states
__global__ void get_field_states(const float *d_states, float *
    h_field_states)
{
  const int thread_ind = blockIdx.x*blockDim.x + threadIdx.x;
  const int states_offset = thread_ind*19;
  const int field_states_offset = thread_ind*1;
  h_field_states[field_states_offset + 0] =
      d_states[states_offset + 17]; //V;
}
```

`set_field_states` is generated in an almost identical manner by simply exchanging the left-hand-side with the right-hand-side in the assignment operations.

8.5.4 Combining the kernels

`CUDACodeGenerator` contains a method to generate all the necessary CUDA kernels at once and combine them into a single string:

Python code

```
def solver_code(self, ode, solver_type):
    code_list = list(
        "// Gotran generated CUDA solver code " \
        "for the \"{0}\" model".format(ode.name))
    code_list.append()
    code_list.append(self.function_code(
        get_solver_fn(solver_type)(ode,
            params=self.params.code)))
    code_list.append(self.init_states_code(ode))
    code_list.append(self.field_states_getter_code(ode))
    code_list.append(self.field_states_setter_code(ode))
    code_list.append(self.init_field_parameters_code(ode))
    return "\n\n".join(code_list))
```

`get_solver_fn` retrieves a function that initialises a Gotran solver component for the specified solver algorithm. This solver component is the `comp` argument used in `function_code` as described in Section 8.5.1.

8.6 Overview of the PyCUDA interface implementation

A layer of abstraction for solving ODEs on the GPU through Python has been implemented as a `CUDAODESystemSolver` class and an auxiliary `ODECUDAHandler` class. They have been integrated with GOSS along with its existing CPU-based solvers that utilise Gotran functionality. The purpose of these classes is to provide a simple interface for the user to solve an ODE system through an interactive Python shell or script, or as part of a larger library.

The `ODECUDAHandler` manages all communication with the GPU, while `CUDAODESystemSolver` provides the user-facing interface.

8.7 `ODECUDAHandler`

The `ODECUDAHandler` class primarily manages the interaction with the CUDA framework through the PyCUDA library. As such, it is a wrapper of PyCUDA functionality for our purposes. This includes initialising the device, initialising and destroying memory on the device, copying data between the host and device and launching CUDA kernels on the GPU.

8.7.1 Initialisation

The CUDA initialisation is done through the `init_cuda` method. This method is called by the `CUDAODESystemSolver` class, and works in two primary stages.

The first stage is the generation and compilation of the CUDA code generated by the `CUDACodeGenerator` described above:

Python code

```
ccg = CUDACodeGenerator(self.params)
self._cuda_code = ccg.solver_code(
    self._ode, self.params.solver)

self.ctx = pycuda.autoinit.device.make_context()
dev = self.ctx.get_device()

nvcc = self.params.nvcc or "nvcc"
gpu_arch = self.params.gpu_arch if self.params.gpu_arch \
           else None
gpu_code = self.params.gpu_code if self.params.gpu_code \
           else None
cuda_cache_dir = self.params.cuda_cache_dir \
                 if self.params.cuda_cache_dir else None

nvcc_options = self.params.nvcc_options
if nvcc_options is not None and len(nvcc_options) > 0 \
        and nvcc_options[0] == "":
    nvcc_options = None

self._mod = SourceModule(
    self._cuda_code, nvcc=nvcc, options=nvcc_options,
    keep=self.params.keep_cuda_code, no_extern_c=False,
    arch=gpu_arch, code=gpu_code, cache_dir=cuda_cache_dir,
    include_dirs=[])
```

```
self.ctx.set_cache_config(pycuda.driver.func_cache.PREFER_L1)
```

First, CUDA code with the specified code generation parameters is generated. Then a new CUDA context and a device object for the GPU is initialised. CUDA contexts are GPU equivalents of CPU processes, while the device object acts as an accessor to the GPU device. The code is then compiled through the PyCUDA `SourceModule` class, and the compiled module is stored.

When compilation is complete, the host and device memory for the ODE system's states, parameters, field states and field parameters is initialised:

Python code

```
# Allocate and initialise states
init_states_fn = self._mod.get_function('init_state_values')
self._h_states = np.zeros(self._num_nodes*self._ode.num_states,
                          dtype=float_t)
self._d_states = pycuda.driver.mem_alloc(
    float_sz*self._num_nodes*self._ode.num_states)
field_states = self.params.code.states.field_states
if len(field_states) == 1 and field_states[0] == "":
    field_states = list()
self._d_field_states = None
if len(field_states) > 0:
    self._d_field_states = \
        pycuda.driver.mem_alloc(
            float_sz*self._num_nodes*len(field_states))
init_states_fn(self._d_states, block=self._get_block(),
               grid=self._get_grid())
pycuda.driver.memcpy_dtoh(self._h_states, self._d_states)
```

Both states and field states have host (CPU) and device (GPU) equivalents to facilitate the transfer of data between the two. After allocating the memory on GPU to hold the state data and setting up the equivalent NumPy arrays on host, the states on the GPU are initialised through `init_states_fn`, which is a PyCUDA function reference to the compiled CUDA kernel on the GPU that performs the data initialisation. The function is called with the NumPy array containing the device (GPU) states, which it then populates. Also note the `block` and `grid` keyword arguments, which are explained in Section 8.7.4.

The initialisation of parameters and field parameters is similar to the states and field states, and the code for this has been omitted.

Finally, we retrieve a PyCUDA reference to the solver function kernel on the GPU and set a flag to mark initialisation as complete:

Python code

```
# Set forward solver function
solver_type = self.params.solver
solver_function_name = \
    self.params.solvers[solver_type].function_name
self._forward_fn = \
    self._mod.get_function(solver_function_name)

self._cuda_ready = True
```

8.7.2 Calling the forward function

The `ODECUDAHandler`'s `forward` function makes the actual call to the forward function on GPU through the PyCUDA wrapper, progressing the computation of the solution to the ODE system on each node by one time step.

Python code

```
def forward(self, t, dt, update_host_states=False,
            synchronize=True):
    """Solve one time step of the ODE system on GPU"""
    if not self.is_ready():
        raise Exception('CUDA has not been initialised')
    else:
        timer = Timer("calculate CUDA forward")

        # Collect the arguments for the CUDA kernel
        args = [self._d_states, t, dt, self._d_parameters]
        field_parameters =
            self.params.code.parameters.field_parameters

        if not (len(field_parameters)==0 or
                (len(field_parameters) == 1
                     and field_parameters[0] == "")):
            args.append(self._d_field_parameters)
        args.append(np.uint32(self.num_nodes))

        # Perform the call to the CUDA kernel
        self._forward_fn(*args,
                         block=self._get_block(),
                         grid=self._get_grid())
```

```
        if synchronize:
            self.ctx.synchronize()
        if update_host_states:
            timer = Timer("update host states")
            pycuda.driver.memcpy_dtoh(self._h_states,
                                      self._d_states)
```

The required arguments for the forward kernel on the GPU as defined by the generated code are collected, and fed into the PyCUDA reference to that kernel. These required arguments are the device (GPU) states `_d_states`, the current time value `t`, the time step `dt`, the device parameters `_d_parameters`, the device field parameters `_d_field_parameters` if any, and the number of nodes `num_nodes`, in that order.

After calling the forward function on the GPU, we may optionally let the GPU threads synchronise, based on the user-supplied `synchronize` flag. This halts the Python program flow until all activity in the current context ceases before continuing. Without synchronisation, the Python script will continue while the forward function runs on the GPU until it returns, or until our next interaction with the GPU is encountered. Synchronisation is thus necessary to accurately measure the runtime of the forward function.

If the `update_host_states` flag is set, the current states are copied from device to host after a call to forward has completed. The user may find this useful for debugging or analysis purposes, at a potential minor cost to runtime efficiency.

8.7.3 Updating field states and field parameters

Setting the field parameters is trivial:

Python code

```
def set_field_parameters(self, h_field_parameters):
    """Copy field states from host to device memory."""
    if not self.is_ready():
        raise Exception('CUDA has not been initialised')
    else:
        pycuda.driver.memcpy_htod(self._d_field_parameters,
                                  h_field_parameters)
```

This simply copies the user-supplied NumPy array containing the desired field parameter values from the host into the device array to be used on GPU, using PyCUDA's `memcpy_htod` function.

The functions to set and get field states require a call to their respective GPU kernel function, but remain relatively simple:

Python code

```
def get_field_states(self, h_field_states):
    """Copy field states from device to host memory."""
    if not self.is_ready():
        raise Exception('CUDA has not been initialised')
    else:
        get_field_states_fn = \
            self._mod.get_function('get_field_states')
        timer = Timer("get_fs_fn")
        get_field_states_fn(self._d_states,
                            self._d_field_states,
                            block=self._get_block(),
                            grid=self._get_grid())
        timer = Timer("get_fs_cpy")
        cuda.memcpy_dtoh(h_field_states,
            self._d_field_states)
```

The function for setting the field states copies them from host to device before the GPU function call, instead of from device to host after the call, but is otherwise identical to its counterpart.

8.7.4 Block and grid size

We pass the number of threads per block and the total number of thread-blocks to the CUDA kernels upon invoking them. These values are calculated by the `_get_block` and `_get_grid` functions, respectively.

Python code

```
def _get_block(self):
    return (min(self._num_nodes, self.params.block_size),
            1, 1)

def _get_grid(self):
    block_size = self.params.block_size
    grid = (self._num_nodes//block_size +
                (0 if self._num_nodes % block_size == 0
                   else 1),
            1)
    return grid
```

The number of threads per block is primarily determined by a user-specified solver parameter, `block_size`, unless it is greater than the total number of nodes. The grid size – the total number of thread-blocks – is then calculated to fit the total number of nodes.

As ODE calculations are not inherently organised in two or three dimensions, but the blocks and the grids are structured in one dimension for the sake of simplicity.

See Section 4.1.1 for more information about CUDA blocks and grids.

8.7.5 Memory and context clean-up

Upon completion, a clean-up function can be called to free up the allocated memory and the CUDA context on the current device:

Python code

```
def clean_up(self):
    """Free the allocated memory and the current device
       context."""
    for _d_array in \
            (self._d_states, self._d_parameters,
             self._d_field_states, self._d_field_parameters):
        try:
            _d_array.free()
        except cuda.LogicError:
            continue
        except AttributeError:
            continue

    if self._cuda_ready:
        self.ctx.pop()
    self._cuda_ready = False
```

8.8 `CUDAODESystemSolver`

The `CUDAODESystemSolver` class is the primary solver interface. It is initialised with the number of nodes in the ODE system, a Gotran `ODE` object describing it together with user-supplied options. These options contain directives important for generating the code and running the ODE solver, such as the floating-point precision, the solver algorithm, the specified field states and field parameters, compiler options and more. They are described in detail in Sections 8.9 and 8.10.

Upon initialisation, the `CUDAODESystemSolver` object performs the following steps:

- If host field states or host field parameters have been supplied, create one local NumPy array for each to store their values on the host.
- Initialise the `ODECUDAHandler` with the given ODE and code generation options. This process is explained in detail in Section 8.7.

After initialising an instance of the class, the user may populate the field parameters on the GPU and retrieve initial field state values from the GPU using the `set_field_parameters` and `get_field_states` methods:

Python code

```
def set_field_parameters(self, field_parameters):
    self.field_parameters = field_parameters
    self._cudahandler.set_field_parameters(
        self.field_parameters)

def get_field_states(self, field_states=None):
    timer = Timer("get field states")
    field_states = field_states if field_states is not None \
                   else self.field_states
    if field_states is not None:
        self._cudahandler.get_field_states(field_states)
```

The main part of the `CUDAODESystemSolver` class is the `forward` function, which computes one time step of the ODE solver algorithm.

Python code

```
def forward(self, t, dt, update_host_states=False):
    """Compute one step of the ODE system."""
    float_t = get_np_float_type(self.params.code)
    t = float_t(t)
    dt = float_t(dt)
    ldt_0 = float_t(self.params.ldt)
    nsteps = int(np.ceil(dt/ldt_0 - 1.0E-12)) \
             if ldt_0 > 0 else 1
    ldt = dt/float_t(nsteps)

    for _ in xrange(nsteps):
        self._cudahandler.forward(t, ldt, update_host_states)
        t += ldt
```

The supplied time and time step values are converted to the appropriate NumPy floating-point type depending on the floating-point precision

specified in the code generation parameters.

The forward function is then called on the GPU through the `ODECUDAHandler`. Note that it is possible to compute multiple time steps per call to this function through a user-supplied substepping value, seen in the above code as `self.params.ldt`. The details of this functionality are described in Section 8.10.

The `forward` method in `CUDAODESystemSolver` is called repeatedly by the user until the ODE system has been simulated for a sufficient length of time.

The `CUDAODESystemSolver` also contains some supplementary methods the user may find useful for debugging or analysis of results:

`get_cuda_states`: Dumps the current states in GPU memory into a NumPy array on the host and returns the array.

`get_cuda_parameters`: Dumps the current parameters in GPU memory into a NumPy array on the host and returns the array.

`get_cuda_code`: Returns a string containing the generated CUDA code used by the solver.

8.9 CUDA code generation parameters

In Gotran, code generation is controlled by a set of user-specified code generation parameters. For the CUDA code generator, typical usage is as follows:

Python code

```python
from gotran import CUDACodeGenerator, load_ode

# Load an ODE from a Gotran form file
ode = load_ode('ode_model.ode')

# Get the default code generation parameters
params = CUDACodeGenerator.default_parameters()

# Update parameters as necessary
params.code.float_precision = "single"
params.code.n_nodes = 1024*1024
params.code.states.field_states = ["V"]

# Generate the CUDA code
cuda_code = CUDACodeGenerator(ode, params)
```

This section details a subset of the supported code generation parameters for the CUDA code generator:

`code.float_precision` specifies the floating point precision of values used for calculation. Accepted values are `"single"` and `"double"` for single- and double-precision floating point computation, respectively. Single-precision floating point operations are significantly faster than double-precision at the expense of accuracy. Section 9.2 details the differences.

`code.n_nodes` sets the number of nodes used in the simulation. The ODE model will be solved for each node in a separate thread. Increasing the number of nodes increases the resolution of the simulation results at increased computation time. The relationship between the number of nodes and simulation runtime is described in Section 9.4.

`code.parameters.field_parameters` supplies the ODE parameter names that will be marked as field parameters, supplied as a list of strings of parameter names. The reader may refer to Section 8.4 for a description of field parameters and field states.

`code.parameters.representation` controls how ODE parameters are represented in the code. Accepted values are `"named"`, `"array"` and `"numerals"`. With the default *named* representation, ODE parameter values are assigned to separate local variables in the CUDA kernel, to be stored in each executing thread's local registers or local memory. With an *array representation*, ODE parameters are instead fetched directly from the global array passed into the kernel by the host. With a *numeric representation*, all references to ODE parameters in the kernel are replaced with their constant numeric values.

`code.states.field_states` and `code.states.representation` are state equivalents to the ODE parameter counterparts. Note that numeric representation for states is not supported, as state values are expected to change throughout the computation unlike static ODE parameters.

`code.body.use_cse` , if set, attempts to optimise the kernel body by using SymPy to extract common sub-expressions.

`code.body.representation` is similar to the `representation` option for ODE states and parameters, but controls the representation of intermediate expressions of component groups in the kernel body. Accepted values are `"named"`, `"array"` and `"reused_array"`. A *named representation* stores each expression in a separate variable in the

executing thread's local registers or local memory. An *array representation* stores them in a local indexed array, while a *reused array representation* will shorten the array and reuse array elements for new expressions when the existing values are no longer needed.

8.10 Solver-specific parameters

`CUDAODESystemSolver` handles solver-specific parameters in a similar manner. It also accepts code generation parameters, which are passed on to the `CUDACODEGenerator` used to generate code for the solver. Sample usage is shown in Section 8.11.

This section details a subset of the supported parameters specific to the solver:

`solver` specifies the algorithm that will be used to solve the ODE system. For this thesis, the explicit Euler, the simplified implicit Euler, the Rush-Larsen and the generalised Rush-Larsen algorithms are supported. These are supplied as `"explicit_euler"`, `"simplified_implicit_euler"`, `"rush_larsen"` and `"generalized_rush_larsen"`, respectively. Brief descriptions of the ODE solver algorithms can be found under Section 3, while test results for the algorithms are detailed in Section 9.6.

`block_size` specifies the number of threads per block with which the CUDA kernel will be executed. See Section 4.1.1 for a description of CUDA threads and blocks. Test results for different values are found in Section 9.3.

`ldt` sets the local time step for the ODE computations. With a local time step lower than the general time step specified for the ODE computation, the ODE's forward function will be called $\lceil dt/ldt \rceil$ times for each iteration, where dt is the general and ldt is the local time step. This is useful when the user supplements the ODE calculation with other operations on field states and field parameters between iterations, and the ODE computations require a higher resolution than the secondary operations. Each iteration will then progress the simulation by the general time step, while the computation of the ODE system undergoes multiple subiterations. If $ldt \leq 0$, substepping is disabled.

`nvcc_options` specifies additional arguments to `nvcc`, supplied as a list of

strings. As an example, to enable the CUDA fast math library for faster, less accurate single-precision floating point computations, the user may set this parameter to `['-ftz=true', '-prec-div=false', '-prec-sqrt=false']`. For complete documentation on the `nvcc` compiler, the reader may refer to [NVIDIA Corporation, 2014b].

8.11 Example solver usage

One example of usage is in the following code:

Python code

```
from goss.cuda import CUDAODESystemSolver
from gotran import load_ode
import numpy as np

# Load the ODE model
ode = load_ode("tentusscher_panfilov_2006_M_cell.ode")

# Get default code generation parameters
params = CUDAODESystemSolver.default_params()

# Set user-specified code generation and solver parameters
params.code.states.field_states = ["V"]
params.code.parameters.field_parameters = ["g_to"]
params.solver = "rush_larsen"
params.block_size = 128

# Set number of nodes
num_nodes = 1024*1024

# Solver keeps all memory and all logic for calling GPU
solver = CUDAODESystemSolver(num_nodes, ode, params)

# Initialise memory for voltage (1 field state)
voltage = np.zeros(num_nodes, dtype=np.float64)

# Initialise memory for 1 field parameter (g_to) as a
# linear transform
g_to_0 = 0.294
field_parameters = \
    (np.arange(num_nodes, dtype=np.float64)+1)*g_to_0/num_nodes

# Load our transformed field parameters from CPU
# onto GPU
```

```
solver.set_field_parameters(field_parameters)

# Load initial ODE state values from GPU into
# our field states on CPU
solver.get_field_states(voltage)

# Do time stepping (in milliseconds)
dt = 0.05
tstop = 25.0

t = 0.0
while t < tstop:
    # Solve ODE
    solver.set_field_states(voltage) # Send to GPU
    solver.forward(t, dt) # Compute one time step of ODE
    solver.get_field_states(voltage) # Get from GPU

    # Do calculations with voltage (solve PDE)...
    voltage[:] = values_from_pde

    # Do additional operations for this time step as
    # necessary here, e.g. plotting a graph of the
    # current field states.

    # Update time
    t += dt
```

In this code example, we load a Ten Tusscher-Panfilov ODE model from a Gotran form file into the object `ode`. We set our initial configuration parameters in `params` and set our solver type as the Rush-Larsen algorithm (see Section 3.1.5). We specify the number of ODEs/nodes to solve in the ODE system with `num_nodes`, and initialise the field states and field parameters that will vary with each node. Finally, we run the simulation for 25 ms with a time step of 50 µs, allowing for additional calculations of the current states at each time step.

Part III
Analysis

9. Benchmarking and test results of GPU acceleration

This section details and discusses the effects of various parameters on simulation runtime and results. We are interested in these results to determine the effects of each parameter on the simulation time and the accuracy of our computations.

Table 9.1: *Key CPU specifications*[13] [14] [15] [16]

Metric	Xeon E5-2687W	Core i7-3632QM
CPU architecture	Sandy Bridge	Ivy Bridge
Process size	32 nm	22 nm
Transistor count	2270 million	1400 million
Cores	8	4
Threads	16	8
Base clock speed	3.1 GHz	2.2 GHz
Turbo clock speed	3.8 GHz	3.2 GHz
L1 cache size	384 kB	256 kB
L2 cache size	2 MB	1 MB
L3 cache size	20 MB	6 MB
Bus architecture	QPI	DPI
Bus transfer rate	8000 MT/s	5000 MT/s
Peak performance (Base)	198.4 GFLOPS	70.4 GFLOPS
Peak performance (Turbo)	243 GFLOPS	102 GFLOPS
Memory bandwidth	Up to 51200 MB/s	Up to 12800 MB/s

We have used two machines to generate simulation results in this section.

[13] <http://ark.intel.com/products/64582/Intel-Xeon-Processor-E5-2687W-20M-Cache-3_10-GHz-8_00-GTs-Intel-QPI>

[14] <http://ark.intel.com/products/71670/Intel-Core-i7-3632QM-Processor-6M-Cache-up-to-3_20-GHz-BGA>

[15] <http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf>

One of the machines contains two Intel Xeon E5-2687W CPUs, one Nvidia GeForce GTX TITAN GPU and 126 GB of RAM. The second consumer-grade machine contains one Intel Core i7-3632QM CPU, one Nvidia GeForce GT 650M GPU and 8 GB of RAM. While we are primarily testing results from these GPUs, some CPU-based simulations are also included for comparison. Section 4.2 provides detailed specifications of the GPUs, while Table 9.1 provides specifications of the CPUs.

9.1 Overview

Table 9.2: *Base test parameters*

Test parameter	Default value
ODE model	Ten Tusscher-Panfilov model
Field states	V (transmembrane potential)
Field parameters	g_{to} (transient outward current)
Field parameter transform	Linear
Number of nodes	65536
Time step	0.1 ms
Simulation time	300 ms
Floating point precision	`double`
Threads per CUDA block	256
Solver algorithm	Rush-Larsen

Table 9.2 shows the base test parameters that are assumed for each simulation in the remainder of this section unless otherwise specified.

The Ten Tusscher-Panfilov model's transmembrane potential ("V") field state is the main variable we have tracked throughout most simulations. It represents the voltage difference across the cell membrane in millivolts. The base conductance of the transient outward current is the main field parameter we will vary across the nodes. It will be transformed linearly, mapping it to 100 % of its original value at the first node, and 1 % at the final node.

See Section 8.4 for details on field parameters, and Figure 8.1 for the effect the linear g_{to} transform has on the Ten Tusscher-Panfilov transmem-

[16] <http://download.intel.com/support/processors/corei7/sb/core_i7-3600_m.pdf>

brane potential. See Section 14.1 for the full Gotran form file of the Ten Tusscher-Panfilov model [Ten Tusscher and Panfilov, 2006].

Table 9.3: *Runtime statistics for 64 identical tests using Nvidia GeForce GTX TITAN*

Metric	Value
Max runtime	7.92 s
Min runtime	7.81 s
Mean runtime	7.84 s
Standard deviation	0.042 s
Mean absolute deviation	0.036 s

Table 9.3 shows a brief summary of basic statistical metrics for 64 identical tests using the base parameters. We can see that the results remain consistent with negligible variations in runtime.

9.2 Floating point precision and fast math

Table 9.4: *Simulation runtime vs. float precision and fast math with* 262144 *nodes*

(a) *Using Nvidia GeForce GTX TITAN*

Float precision	Use fast math	Time (s)
Single	No	6.37
Single	Yes	3.82
Double	No	30.1
Double	Yes	30.1

(b) *Using Nvidia GeForce GT 650M*

Float precision	Use fast math	Time (s)
Single	No	40.3
Single	Yes	24.9
Double	No	165
Double	Yes	165

The computation of solutions to ODE systems are done with floating point numbers, as they are the standard method of approximating real numbers

in computing. Floating point numbers are typically represented with either single or double precision, and Gotran supports generating C code for both representations. As single-precision floating point numbers store the approximation with fewer bits than double-precision floats, they are typically faster to process at the expense of accuracy. As such, it may be interesting to quantify both the speed-up we can achieve and the precision we sacrifice by using single-precision calculations.

Additionally, CUDA supports compilation with optimisations of certain mathematical operations involving single-precision floating point numbers at further expense of accuracy. These optimisations are done through the CUDA fast math library, which approximates specific mathematical operations and floating point representations, instead of using IEEE-compliant operations [NVIDIA Corporation, 2014a]. Specifically, CUDA's fast math library replaces single-precision division and square root operations with faster approximations, and flushes floating point denormals[17] to zero.

Table 9.4 illustrates how float precision and usage of CUDA's fast math library affect simulation runtime. The fast math library was included by compiling the generated CUDA source code with `nvcc` parameters `-ftz=true`, `-prec-div=false` and `-prec-sqrt=false`. Note especially that the fast math library has no effect on double-precision runtime. This is consistent with the CUDA documentation, which specifies that the fast math library only applies to single-precision operations [NVIDIA Corporation, 2014a].

There is a significant speed-up from utilising single-precision over double-precision floating point numbers. For the Nvidia GeForce GTX TITAN, this speed-up is almost by a factor of five. Enabling the fast math library for single-precision floats brings down the runtime further by a factor of 1.6, with a total speed-up at almost eight times compared to double-precision floats.

9.2.1 Floating point precision accuracy

In addition to the speed-ups, it is also important to determine how much single-precision calculations affect the accuracy of the solution.

[17] Floating point values below the lowest possible normalised value with the lowest representable exponent, where the floating point representation is denormalised by setting the leading binary significand to zero. Allows representing numbers closer to zero at some cost to performance.

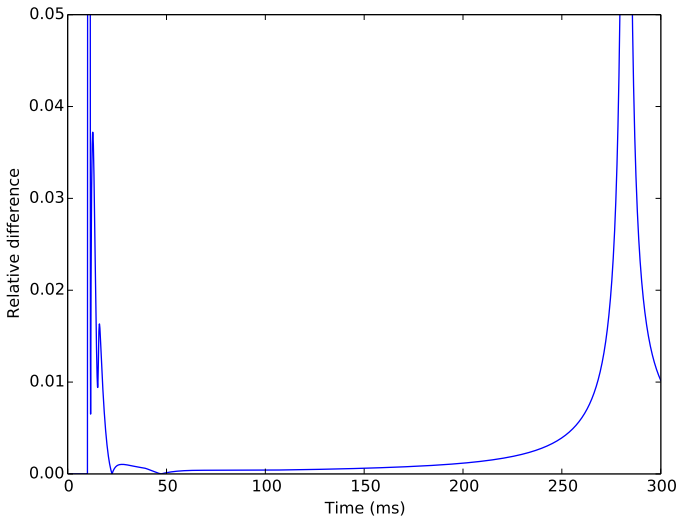


Figure 9.1: *Relative difference between single- and double-precision floating point calculations of the membrane potential of the Ten Tusscher-Panfilov model*

Figure 9.1 shows the relative difference between single- and double-precision floating point calculations of the membrane potential in the Ten Tusscher-Panfilov model. The relative difference δ_t at each time step t was calculated with double-precision floats as reference, as follows:

$$\delta_t = \left| \frac{f_t - d_t}{d_t} \right| \tag{9.1}$$

where f_t and d_t are the membrane potential at time step t for floats and doubles, respectively.

Note the spikes in relative difference around 11 ms and 280 ms. There are two contributing factors to these spikes. First, at approximately 10 ms the depolarisation phase of the action potential, causing a rapid change in transmembrane potential, is triggered a few milliseconds earlier by the single-precision floating point simulation than the double-precision simulation. Secondly, at both 11 ms and 280 ms, the calculated membrane potentials switch polarity as they cross 0 mV. As d_t approaches 0, δ_t approaches the asymptote $\left| \frac{f_t}{0} \right|$.

This second factor is misleading in these calculations, as the asymptotic increase in relative difference when the denominator approaches zero is not representative of the small differences in accuracy we expect between

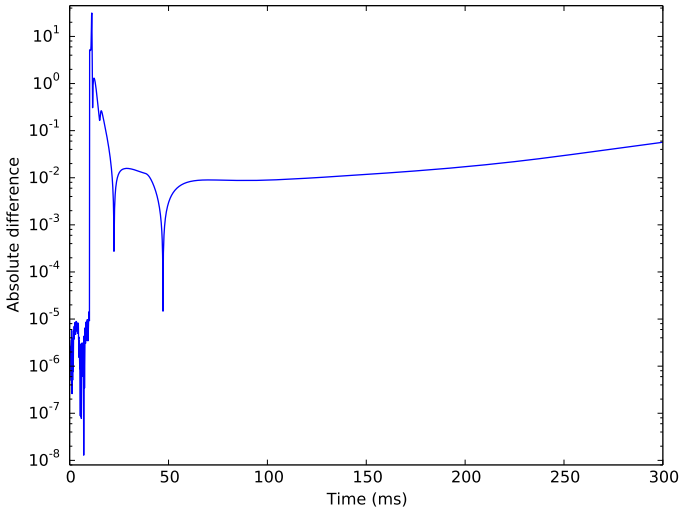


Figure 9.2: *Absolute difference between single- and double-precision floating point calculations of the membrane potential of the Ten Tusscher-Panfilov model (log-lin)*

floating point precisions.

A more representative metric may be the absolute difference between the single- and double-precision floating point calculations, calculated as follows:

$$\delta_t = |f_t - d_t| \tag{9.2}$$

With this metric, illustrated on our data in Figure 9.2, the effect of the unsynchronised activation of the depolarisation phase is still apparent around 11 ms when the absolute difference in calculated voltage briefly reaches approximately 34 mV. The noise from switching polarity by crossing 0 mV, however, is eliminated. After the depolarisation phase, the absolute difference appears to stabilise around 0.01 mV before slowly increasing to 0.06 mV at 300 ms. The more complex Grandi-Pasqualini-Bers model exhibits a comparable result in absolute difference between floating point precisions. This is shown in Figure 9.3, where the membrane potential for this model has been computed with the generalised Rush-Larsen algorithm at a time step of 12.5 µs. The peak difference is again around the depolarisation phase, which occurs around 4 ms with the Grandi-Pasqualini-Bers simulation.

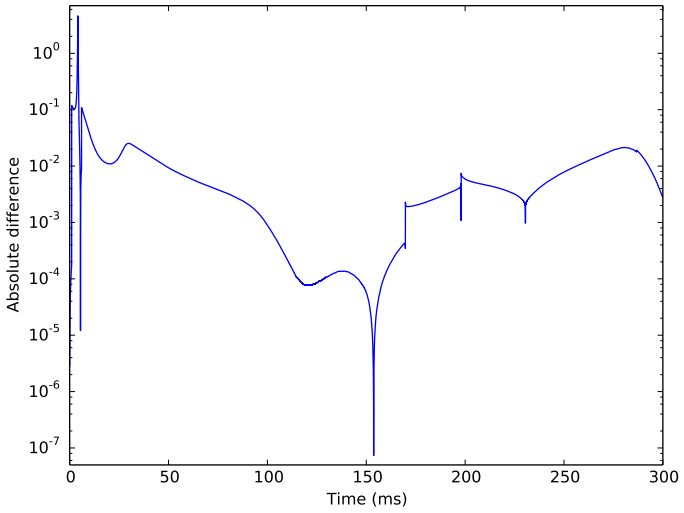


Figure 9.3: *Absolute difference between single- and double-precision floating point calculations of the membrane potential of the Grandi-Pasqualini-Bers model (log-lin)*

For a closer look at the depolarisation inconsistency, refer to Figure 9.4 which shows the transmembrane potential for the single- and double-precision floating point simulations of the Ten Tusscher-Panfilov model around 11 ms. Note that the difference in the time of onset for the depolarisation phase is 0.1 ms, at 10.0 ms and 10.1 ms for the single- and double-precision calculations, respectively. This is exactly one time step for our Ten Tusscher-Panfilov computation. Recomputing this model with a time step of 12.5 µs also reduces the difference in time of onset to 12.5 µs between the single- and double-precision calculations, suggesting that this source of inconsistency may be insignificant for simulations with a low time-step.

9.2.2 Fast math accuracy

For the sake of completeness, we also looked at the effect of the fast math library on the accuracy of calculations. The absolute difference in transmembrane potential for the Ten Tusscher-Panfilov model with single-precision floating point calculations with and without the fast math library is shown in Figure 9.5. As this difference consistently stays under 12 nV, we may consider it insignificant for computations with our parameters.

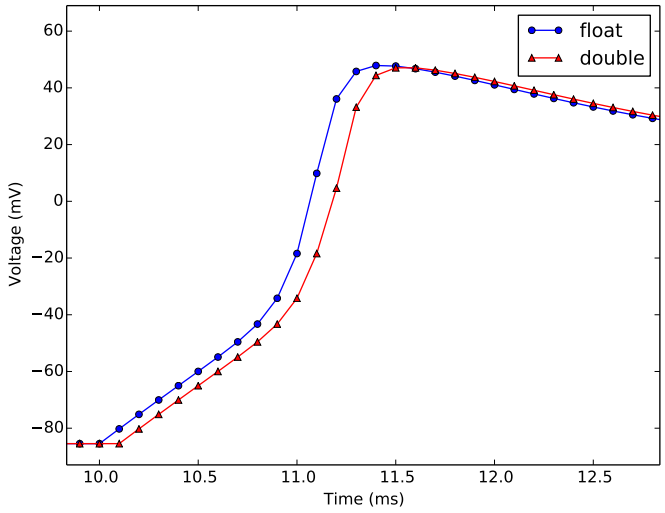


Figure 9.4: *Depolarisation phase of the Ten Tusscher-Panfilov model, simulated with single and double-precision floating point calculations*

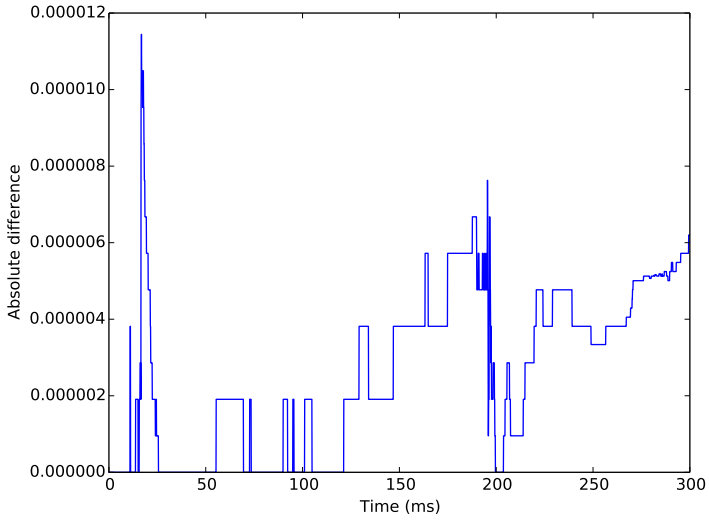


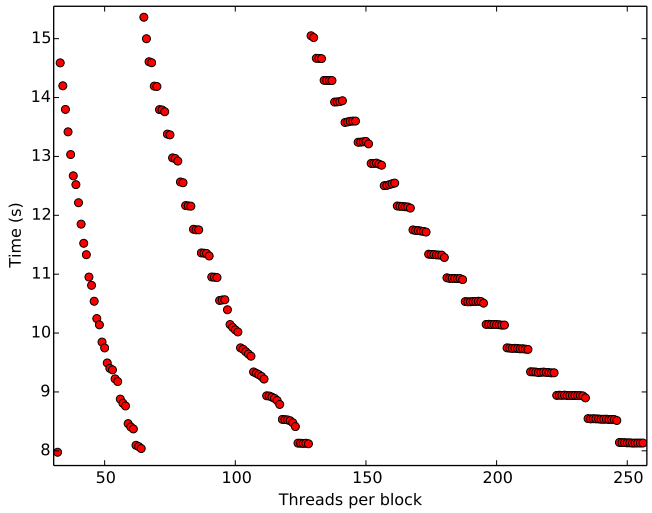
Figure 9.5: *Absolute difference between calculations of the membrane potential of the Ten Tusscher-Panfilov model with and without fast math*

9.3 Threads per block

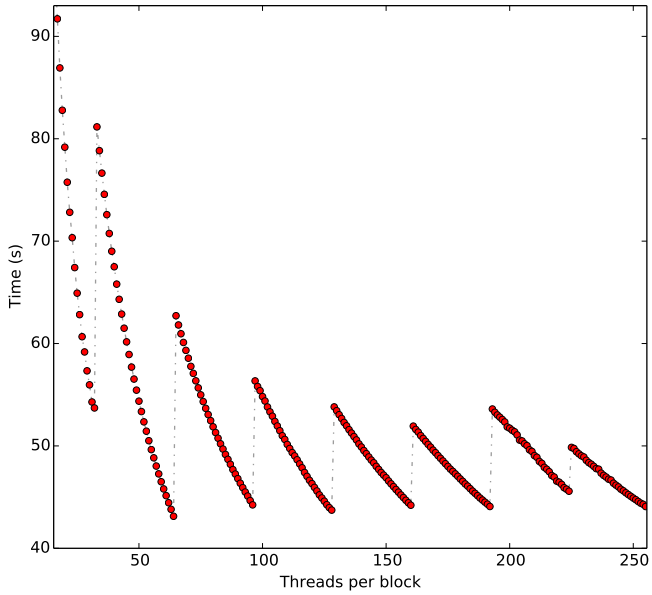
Figure 9.6 shows the impact of the block size on the runtime of an ODE simulation. The simulations used the Rush-Larsen algorithm to simulate the Ten Tusscher-Panfilov model over 300 ms at 100 µs increments, with 65536 nodes per time step.

As described in Section 4.1.3, threads are executed in parallel in groups of 32 known as warps. This applies for both the Kepler microarchitecture used by the Nvidia GeForce GTX TITAN and the older Fermi microarchitecture used by the Nvidia GeForce GT 650M.

We would expect the simulations to reach peak efficiency at multiples of this warp size, which matches the results from the consumer-grade GeForce GT 650M. Note, however, that the results from the GeForce GTX Titan do not match these predictions. For the GTX Titan, peak efficiency appears to occur at powers of two, e.g. at 32, 64, 128 and 256 threads per block for the values tested. Further research would be needed to determine what causes these results.



(a) *Using Nvidia GeForce GTX TITAN*



(b) *Using Nvidia GeForce GT 650M*

Figure 9.6: *Simulation runtime for varying number of threads per block*

9.4 Number of nodes

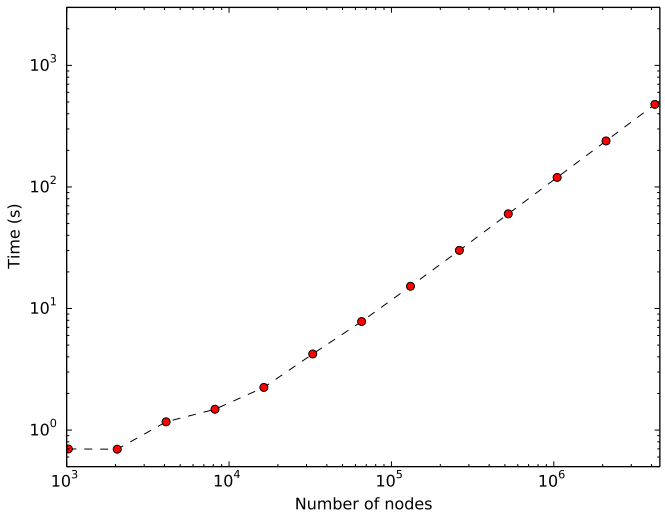
Table 9.5: *Runtime speed-up from GT 650M to GTX TITAN for number of nodes*

N_{nodes}	GT 650M time (s)	GTX TITAN time (s)	Speed-up
1024	1.136	0.699	1.6 ×
2048	1.727	0.696	2.5 ×
4096	3.111	1.168	2.7 ×
8192	5.822	1.483	3.9 ×
16384	11.13	2.243	5.0 ×
32768	21.49	4.228	5.1 ×
65536	42.10	7.816	5.4 ×
131072	83.21	15.24	5.5 ×
262144	164.8	30.08	5.5 ×
524288	327.8	60.07	5.5 ×
1048576	653.8	119.7	5.5 ×
2097152	1306	239.0	5.5 ×
4194304	2609	477.9	5.5 ×

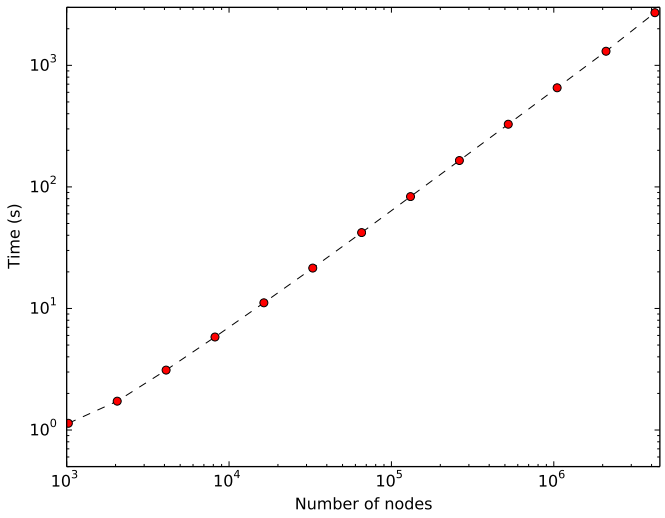
The number of simulated nodes controls the resolution at which the model is calculated. Each iteration of the ODE system is computed once for each node, typically with minor variations in select field parameters across the nodes. Table 9.5 shows how the number of simultaneously simulated nodes affect the runtime of a simulation. The runtimes are also plotted in Figure 9.7 as log-log plots for a visual overview.

The number of nodes was doubled for each subsequent simulation, from 1024 at the lower end to a maximum of 4194304 nodes. As each node adds one more ODE system to be solved, the simulation runtime appears to be proportional to the number of simulated nodes. Simple linear regression models for the runtime data on each GPU both yield an $R^2 > 0.9999$.

As the number of nodes increases to a point where the computational overhead becomes insignificant, the speed difference between our GPUs also stabilises with the Nvidia GeForce GTX TITAN being approximately 5.5 times faster.



(a) *Using Nvidia GeForce GTX TITAN (log-log)*



(b) *Using Nvidia GeForce GT 650M (log-log)*

Figure 9.7: *Simulation runtime vs. number of nodes calculated per time step*

9.5 ODE model

Table 9.6: *Simulation runtime vs. ODE model with 65536 nodes*

(a) *Using Nvidia GeForce GTX TITAN*

ODE Model	States	Parameters	LoC[18]	Time (s)
Beeler-Reuter	8	10	132	23.4
Ten Tusscher-Panfilov	19	53	456	99.5
Grandi-Pasqualini-Bers	39	107	852	298

(b) *Using Nvidia GeForce GT 650M*

ODE Model	States	Parameters	LoC	Time (s)
Beeler-Reuter	8	10	132	172
Ten Tusscher-Panfilov	19	53	456	593
Grandi-Pasqualini-Bers	39	107	852	1745

The complexity of the ODE model also significantly affects simulation time, as shown in Table 9.6. For this comparison, each model was simulated in 12.5 µs increments with the second-order generalised Rush-Larsen solver algorithm. The algorithm and time step were chosen due to the stiffness of the Grandi-Pasqualini-Bers model, which causes instability when calculated with larger time steps than the more basic first-order Rush-Larsen algorithm.

Note especially the relationship between the lines of code in the CUDA forward function and the simulation time for each model. For both GPUs, the average time spent per line of code increases significantly for the Grandi-Pasqualini-Bers model. For the consumer-grade Nvidia GeForce GT 650M GPU, the Beeler-Reuter, Ten Tusscher-Panfilov and Grandi-Pasqualini-Bers models respectively spend an average of approximately 20 µs, 19 µs and 31 µs of real time per line of code per node. For the Nvidia GeForce GTX TITAN, these numbers are 2.7 µs, 3.3 µs and 5.3 µs for each respective model.

[18]Lines of code in the generated CUDA forward function

It is likely that this discrepancy arises from the size and complexity of the Grandi-Pasqualini-Bers model causing register spill on the GPU. During compilation, `nvcc` assigns variables declared in each device function to a limited set of thread-local GPU registers. The Nvidia GeForce GTX TITAN, for instance, has 255 available registers per thread. When the number of declared local variables exceeds this amount, variables will "spill" into local memory, which is supported by the cache hierarchy. Each spilled variable must be loaded and stored on reads and writes, which is significantly slower than accessing it directly from the thread's registers.

Compiling the generated CUDA code with the `nvcc` compiler option `--ptxas-options=-v` displays the amount of used registers and the memory needed per compiled device function. This data shows that the Grandi-Pasqualini-Bers forward function has a significantly higher register spill than the Ten Tusscher-Panfilov forward function, while Beeler-Reuter has none.

The following is an excerpt from the `xptxas` output detailing the register and memory usage by the forward function for the Grandi-Pasqualini-Bers model on the Nvidia GeForce GTX TITAN GPU:

Terminal output

```
ptxas info    : Compiling entry function
    'forward_generalized_rush_larsen' for 'sm_35'
ptxas info    : Function properties for
    forward_generalized_rush_larsen
    1136 bytes stack frame, 1160 bytes spill stores, 3172
        bytes spill loads
ptxas info    : Used 254 registers, 356 bytes cmem[0], 1072
    bytes cmem[2]
```

The output specifies that all available thread registers are used, with 1072 additional bytes per thread being stored in thread-local memory (`cmem[2]`). In each thread, 1160 bytes are stored in spilled memory, while 3172 bytes are loaded from it.

9.6 Solver algorithms

Table 9.7 shows the simulation runtimes for different solver algorithms applied to the Ten Tusscher-Panfilov model using a time step of 12.5 µs. The explicit Euler algorithm is far too unstable to converge upon a solution at this time step with the Ten Tusscher-Panfilov model, and was therefore

Table 9.7: *Simulation runtime vs. solver algorithm with 65536 nodes*

(a) *Using Nvidia GeForce GTX TITAN*

Solver	Time (s)
Rush-Larsen	62.77
Generalised Rush-Larsen	100.5
Simplified implicit Euler	96.83

(b) *Using Nvidia GeForce GT 650M*

Solver	Time (s)
Rush-Larsen	336.8
Generalised Rush-Larsen	592.7
Simplified implicit Euler	568.2

excluded from these tests.

9.6.1 Highest stable time step

To account for some solvers being more stable and accurate than others, we compared a "stable" solution for each solver with a reference solution. The stable solution was chosen such that the average transmembrane potential did not differ more than 1 % from the reference solution.

The reference solution for the Ten Tusscher-Panfilov model was computed with the Rush-Larsen algorithm with a time step of 1 µs. For each algorithm tested against this control, the highest time step that satisfied our constraint was selected. For the stiffer Grandi-Pasqualini-Bers model, the reference solution was calculated with the generalised Rush-Larsen algorithm with a time step of 0.125 µs. The results are shown in Table 9.8.

For the Ten Tusscher-Panfilov model, it is apparent that the explicit Euler algorithm is far too unstable for practical use compared to the other tested algorithms. It requires a time step approximately 170 times lower and a computation time almost 27 times higher than the Rush-Larsen algorithm to reach the same level of accuracy. From the results, the Rush-Larsen algorithm is the most efficient algorithm to solve the Ten Tusscher-Panfilov model by a wide margin.

Interestingly, these drastic differences in the stability of the solver

Table 9.8: *Simulation runtime for solver algorithms at highest stable time step with 65536 nodes using Nvidia GeForce GTX TITAN*

(a) *Using the Ten Tusscher-Panfilov model*

Solver	Time step (µs)	Time (s)
Rush-Larsen	338	2.36
Generalised Rush-Larsen	208	6.02
Simplified implicit Euler	94	13.3
Explicit Euler	2	62.8

(b) *Using the Grandi-Pasqualini-Bers model*

Solver	Time step (µs)	Time (s)
Rush-Larsen	8	197
Generalised Rush-Larsen	10	375
Simplified implicit Euler	6	655
Explicit Euler	7	185

algorithms are significantly lessened for the stiffer Grandi-Pasqualini-Bers model. For this model, the explicit Euler algorithm is able to achieve an equivalent level of stability in the shortest time compared to the other algorithms. The reasons for this are unclear, and may be appropriate to investigate in a future study. It should also be noted that every algorithm other than the generalised Rush-Larsen algorithm rapidly destabilises with increased time steps for this model. Specifically, at 10 µs for Rush-Larsen and simplified implicit Euler, and at 8 µs for explicit Euler, no solution is found as the computation diverges to infinity. This phenomenon can be attributed to the stiffness of the Grandi-Pasqualini-Bers model, where these numerical solvers are constrained primarily by stability concerns rather than accuracy. The generalised Rush-Larsen algorithm, however, will find a solution for the Grandi-Pasqualini-Bers model at higher time steps at a steadily increasing cost to accuracy.

9.7 Field states and field parameters

In this section, we are interested in testing the differences in simulation runtime when field states and field parameters are used. Field states and

Table 9.9: *Simulation runtime vs. field parameters*

(a) *Using Nvidia GeForce GTX TITAN*

Field parameters	Field parameter transform	Time (s)
g_{to}, g_{CaL}	Linear transform	7.94
g_{to}	Linear transform	7.88
$\varnothing$	$\varnothing$	7.91

(b) *Using Nvidia GeForce GT 650M*

Field parameters	Field parameter transform	Time (s)
g_{to}, g_{CaL}	Linear transform	42.4
g_{to}	Linear transform	42.1
$\varnothing$	$\varnothing$	42.2

field parameters are described in detail in Section 8.4.

For these tests, the Ten Tusscher-Panfilov model was used with our base test parameters. The field parameters had their initial values set with a linear transform as described in Section 9.1. They were not updated between the time steps of the simulation. The field states were copied between the host and the device on each time step, but no further calculations were performed on the host.

Tables 9.9 and 9.10 show the results for field parameters and field states, respectively. While we expected a minor performance hit by using field parameters and field states from the increase in memory transfers, the differences in runtimes appear to be negligible. This may be explained by the computationally heavy algorithms solving the ODE on each time step dwarfing the relatively minor memory transfers.

Table 9.10: *Simulation runtime vs. field states*

(a) *Using Nvidia GeForce GTX TITAN*

Field states	Time (second)
V	7.87
$\varnothing$	7.86

(b) *Using Nvidia GeForce GT 650M*

Field states	Time (second)
V	42.1
$\varnothing$	42.0

9.8 GPU vs. CPU solvers

It may be interesting to see the differences in performance between our CUDA implementation for solving ODEs with existing CPU implementations. The tests in this section solve the Ten Tusscher-Panfilov model with our base test parameters on CPU and GPU, with both our consumer-grade and workstation hardware. The multi-threaded CPU-based solver used for these tests was provided by GOSS. The CPU-based solver only supports double-precision floating point calculations, while we tested both double- and single-precision with our GPU-based implementation.

Table 9.11 shows the ODE simulation runtime speed-ups on our different sets of hardware. We see significant speed-ups with our GPU-based implementation for both the consumer-grade and the workstation hardware, especially with the use of single-precision floats. The speed-ups also remain consistent with a large number of simulated nodes, with a moderate improvement for single-precision simulations as the relative impact of overhead is alleviated. This can be seen in Table 9.11b.

Table 9.11: *Simulation runtime on GPU vs. CPU*

(a) *Using* 65536 *nodes*

Processor	Floats	Time (s)	Speed-up
33.547 2x Xeon E5-2687W CPU (16 threads)	`double`	33.5	N/A
GTX TITAN GPU	`double`	9.73	3.4 ×
GTX TITAN GPU	`single`	3.07	10.9 ×
Core i7-3632QM CPU (4 threads)	`double`	137	N/A
GT 650M GPU	`double`	44.0	3.1 ×
GT 650M GPU	`single`	11.3	12.1 ×

(b) *Simulation runtime on GPU vs. CPU with* 4194304 *nodes*

Processor	Floats	Time (s)	Speed-up
2x Xeon E5-2687W CPU (16 threads)	`double`	2000	N/A
GTX TITAN GPU	`double`	556	3.6 ×
GTX TITAN GPU	`single`	137	14.7 ×

9.9 GPU vs. CPU solvers with PDE simulation and OpenMPI parallelism

While we see significant speed-ups for solving ODEs on the GPU in Section 9.8, full simulations of the properties of cellular interactions will also typically involve solving a PDE step, (Equations 2.4 and 2.5) between each ODE step (Equation 2.6). The membrane potential from the PDE is fed back into the ODE system via field states for each time step. It may be interesting to see how the performance of such simulations compares between CPUs and GPUs with various parameters.

The tests in this section were done on a 2D mesh with 2048×2048 nodes for a total of 4194304 nodes over 25 ms. A time step of 125 µs was used with the Rush-Larsen solver on the Ten Tusscher-Panfilov model. Table 9.12 shows the total cumulative runtime for computing ODEs, PDEs and the total runtime of each simulation. Table 9.12c shows the speed-up of simulations on GPU compared to their equivalent computations on CPU. Note that in the single-precision cases, only the ODE was computed as such; the PDEs were solved using double-precision in all cases. Also note

Table 9.12: *Simulation runtime of the monodomain equation with interleaved PDE and ODE simulations*

(a) *Using 2x Intel Xeon CPU E5-2687W 3.10 GHz (double-precision floats)*

MPI processes	ODE time (s)	PDE time (s)	Total time (s)
1	278	1121	1399
4	286	391	677
8	259	242	501
16	375	225	599

(b) *Using Nvidia GeForce GTX TITAN*

MPI processes	Floats	ODE time (s)	PDE time (s)	Total time (s)
1	`single`	36.3	1119	1155
4	`single`	25.2	395	420
8	`single`	24.9	258	283
16	`single`	20.8	211	232
1	`double`	87.9	1117	1205
4	`double`	77.9	394	472
8	`double`	75.4	261	336
16	`double`	74.8	204	279

(c) *Speed-up from CPU to GPU*

MPI processes	Floats	ODE speed-up	Total speed-up
1	`single`	7.6 ×	1.2 ×
4	`single`	11.3 ×	1.6 ×
8	`single`	10.4 ×	1.8 ×
16	`single`	18.0 ×	2.6 ×
1	`double`	3.2 ×	1.2 ×
4	`double`	3.7 ×	1.4 ×
8	`double`	3.4 ×	1.5 ×
16	`double`	5.0 ×	2.1 ×

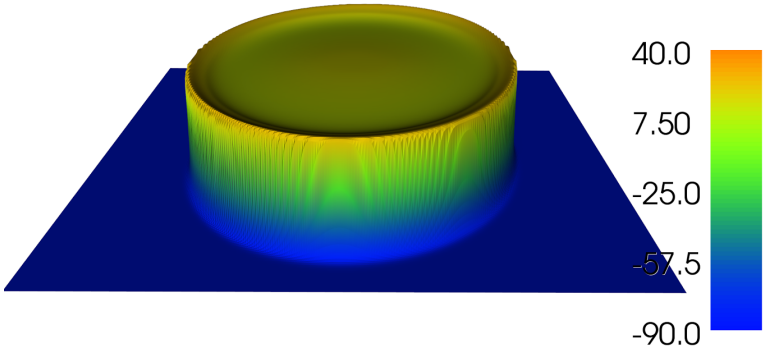


Figure 9.8: *Voltage propagation over 2D plane after* 25 ms *with the Ten Tusscher-Panfilov model*

that the ODE simulations on CPU were run with double-precision floating point operations as there was no option for performing these runs with single precision in the existing GOSS implementation.

The Xeon CPU results (Table 9.12a) were run using a CPU-based solver implementation from GOSS, while the GTX TITAN GPU results (Table 9.12b) utilised this thesis' CUDA implementation of a Gotran solver.

A new parameter introduced in these tests was the number of MPI processes, controlled via tools by the Open MPI Project[19]. MPI is a library specification for message-passing between processes, and allows us to split simulations into separate parallel runs on the host through several MPI processes. This has different effects on the purely CPU-based ODE simulations from GOSS and the GPU-based ODE simulations implemented in this thesis.

For the GPU simulations, each MPI process constructs a separate `CUDAODESystemSolver` that each solves a subset of the ODE system nodes on the GPU simultaneously. It was not immediately apparent that this should work well, as memory conflicts or interferences between instances of CUDA contexts may be expected from multiple host processes attempting to access the GPU at the same time. However, separate CUDA contexts are spawned by each instance of the `ODECUDAHandler` (see Section 8.7.1). These are the GPU equivalents of CPU processes, and the PyCUDA library appears to automatically prevent the contexts from competing over system

[19] `<http://www.open-mpi.org/>`

resources.

From the results, we can see a moderate speed-up for ODE simulations on the GPU based on the number of MPI processes. The execution of the GPU kernel is performed with 256 threads per block. With 4194304 simulated nodes, this results in 16384 total thread-blocks per time step of the computation. These numbers far exceed the GPU's warp size of 32 threads executed on its 2688 cores, suggesting that the GPU is already running at full capacity regardless of the number of processes simultaneously invoking the kernel code. Therefore, we suspect that the performance increase primarily stems from a reduction in CPU overhead, likely during memory transfers. However, further work would be needed to determine a conclusive explanation.

The CPU-based ODE simulations are performed through existing GOSS solvers. These solvers already use OpenMP parallelism[20], not to be confused with MPI. Whereas MPI provides message-passing between processes, OpenMP is a specification for shared memory parallelism in C/C++ programs through library routines and compiler directives. As the number of threads on the CPUs are limited at 8 threads per CPU or 16 threads in total, increasing the number of MPI processes necessitates a reduction in the number of OpenMP threads per MPI process. With one process, all 16 threads are utilised for OpenMP parallelism, while with eight MPI processes, at most two OpenMP threads can be used per process.

For one, four and eight MPI processes, this does not seem to have a significant effect on the runtime for solving the ODE systems, but the result for 16 MPI processes is interesting. With 16 MPI processes and one thread per process, the performance degrades significantly. We may speculate that this is due to the memory and execution processes in threaded runs being better aligned in OpenMP. However, further research is needed to determine the exact cause and whether this effect can be minimised.

The double-precision CPU and GPU simulations and the single-precision GPU simulations all solve the same PDE using the same PDE solver. As such, the time to calculate PDEs remained relatively consistent based on the number of MPI processes, irrespective of other differences in parameters for solving the ODEs. The PDEs were solved numerically using the finite element method as implemented by the FEniCS project[21], and the

[20] <http://openmp.org/wp/>

[21] <http://fenicsproject.org/>

resulting large linear systems were solved using iterative solvers provided by the PETSc library[22].

Figure 9.8 illustrates the resulting voltage over the mesh of nodes, representing the propagation of the action potential across the heart at 25 ms. Note that this figure was generated with a lower resolution than was used for the runtime tests, with a mesh of 128×128 nodes.

Overall, the significant speed-ups seen in the ODE simulations on the GPU over the CPUs are somewhat dwarfed by the large amount of computation time used to serially solve the PDEs.

9.10 PDE simulation with ODE substepping

While Section 9.9 tested a simulation of the Ten Tusscher-Panfilov model with PDE solvers, a stiffer and more complex model such as Grandi-Pasqualini-Bers requires a far lower time step to produce a stable solution. The PDEs to be solved, however, may remain identical. Lowering the time step for the PDEs along with the ODEs may therefore not be necessary.

The simulations in this section were run using the Grandi-Pasqualini-Bers model with 2048×2048 or 4194304 over 25 ms. A PDE time step of 250 µs was used, with a local ODE time step of 12.5 µs. With these values, one iteration of the PDE system was computed for every 20 iterations of the ODE system. The ODE systems were solved with the generalised Rush-Larsen algorithm.

For these tests, we get a comparable relative speed-up from CPU to GPU for ODE computations as the speed-up seen in Section 9.9. With these tests, however, we also see a significant speed-up in total computation time as the PDE calculations are dwarfed by the computationally heavier ODE calculations.

[22] <http://www.mcs.anl.gov/petsc/>

Table 9.13: *Simulation runtime with PDE simulation on every* 20 *ODE steps*

(a) *Using 2x Intel Xeon CPU E5-2687W 3.10 GHz (double-precision floats)*

MPI processes	ODE time (s)	PDE time (s)	Total time (s)
1	5901	578	6479
4	5791	269	6059
8	6095	245	6340
16	7919	122	8041

(b) *Using Nvidia GeForce GTX TITAN*

MPI processes	Floats	ODE time (s)	PDE time (s)	Total time (s)
1	`single`	452	593	1045
4	`single`	471	279	749
8	`single`	495	226	721
16	`single`	501	119	620
1	`double`	1530	610	2140
4	`double`	1578	276	1855
8	`double`	1580	212	1793
16	`double`	1627	123	1749

(c) *Speed-up from CPU to GPU*

MPI processes	Floats	ODE speed-up	Total speed-up
1	`single`	13.1 ×	6.2 ×
4	`single`	12.3 ×	8.1 ×
8	`single`	12.3 ×	8.8 ×
16	`single`	15.8 ×	13.0 ×
1	`double`	3.9 ×	3.0 ×
4	`double`	3.7 ×	3.3 ×
8	`double`	3.9 ×	3.5 ×
16	`double`	4.9 ×	4.6 ×

10. Summary

With the extensions to Gotran implemented in this thesis, we have achieved a number of goals:

- Further automation of the process of working with ordinary differential equations through automatic LaTeX code generation.
- Optimisation of the computationally heavy large-scale simulations of millions of ODEs typically encountered in cardiac modelling.
- Incorporation of the strengths and ideas from existing tools and frameworks to extend a powerful Python framework accessible to computational researchers and scientists.

We have demonstrated that a significant performance boost can be gained by computing highly complex ODE models on high-end GPUs over similarly high-end CPUs. With the improvements to performance for heavy ODE computations, serial PDE calculations may become the bottleneck for full simulations of cellular electrophysiology. Depending on the specific model and simulation parameters, the simulation time spent on calculating PDEs can easily exceed 50 %, and even approach 95 % in specific cases. Despite the PDE bottleneck, stiffer and more complex models such as Grandi-Pasqualini-Bers still see a notable performance benefit when solved with ODE substepping.

The CUDA code generator and the associated solver interface have both been implemented as parts of extensible frameworks and can easily be amended to support new solver algorithms and further optimisations, either through further framework development or through usage by third-party libraries.

We have also successfully built an automated tool for generating LaTeX-documents describing large and complex ODE systems, alleviating the tedious work required to do so by hand. The generated code is aimed at being easily managed and changed to integrate into existing manuscripts with minimal hassle.

Both these tools allow researchers to work more efficiently with cardiac models by alleviating tedious tasks and low-level details.

10.1 Future work

In 2010 Fred Lionetti pioneered a CUDA implementation of an ODE solver similar to what we have implemented [Lionetti, 2010]. In his thesis several ODE solvers were investigated. Lionetti notably battled with low GPU memory size and bandwidth and had to split up right-hand-side evaluations into multiple but smaller parts, avoiding register spilling. This made the code unnessary complex and unfortunately cell model specific. With recent years' improvements to GPU hardware, we anticipated that we would get less register spilling and a lower memory latency when they occur.

With the high-end Nvidia GeForce GTX TITAN, we were able to avoid register spilling entirely for the Ten Tusscher-Panfilov and Beeler-Reuter models, but the Grandi-Pasqualini-Bers model proved too large for our register sizes. While it only spilled approximately a thousand bytes per thread, it would be useful to see if the partitioning of solver kernels would bring a signficant benefit. Kernel splitting would also be useful for lower-end consumer-grade hardware that may experience register spill with smaller models. Alternatively, we anticipate that support for caching data into a thread-block's shared memory also would be advantageous compared to simply allowing variables to spill into thread-local memory, due to the higher bandwidth and lower latency of shared memory [NVIDIA Corporation, 2014a].

Additionally, it would be beneficial to extend our contributions with multi-GPU support. We do not anticipate this to be a significant challenge with the PyCUDA library's management of hardware devices, GPU contexts and CUDA streams, as we were able to launch multiple instances of our GPU solver on separate parallel host processes with our implementation on one GPU.

Part IV
Appendix

11. Acknowledgements

I would like to extend my sincere thanks to Johan Hake (Simula Research Laboratory), the primary developer of the Gotran and GOSS libraries and the main supervisor for this thesis. This thesis could not have been written without his endless help with pointers, resources, proof-reading and theory.

12. Installation

12.1 Core installation of Gotran

Gotran runs on Python 2.7, which can be installed from <https://www.python.org/>.

As of this writing, Gotran depends on the following Python packages:

- Instant[23]
- SymPy[24]
- NumPy[25]
- modelparameters[26]

Clone or download the Gotran git repository[27] and follow the installation instructions in the `INSTALL` file.

12.2 Installation of CUDA and PyCUDA

Gotran uses CUDA to run simulations on GPU. This requires an Nvidia graphics card that supports the CUDA Toolkit. Extensive installation instructions for CUDA are available at Nvidia's "CUDA Zone"[28].

Gotran uses the PyCUDA Python wrapper to access the lower-level CUDA environment directly from Python. PyCUDA is available at <http://mathema.tician.de/software/pycuda/>.

12.3 GOSS

The PyCUDA-based ODE solver interface implemented in this thesis is available in GOSS. GOSS can be downloaded or forked from its git repository[29] with installation instructions in the `INSTALL` file.

[23] <https://launchpad.net/instant>
[24] <http://sympy.org>
[25] <http://numpy.scipy.org>
[26] <https://launchpad.net/modelparameters>
[27] <https://bitbucket.org/johanhake/gotran>
[28] <https://developer.nvidia.com/cuda-zone>
[29] <https://bitbucket.org/johanhake/goss>

12.4 LaTeX generation

While Gotran does not depend on any additional software to generate LaTeX code, additional tools are needed to compile the generated code to a readable format. The generated code is tailored for the TeX Live software distribution on Linux systems, and should be compiled with `pdflatex`.

Installation instructions for TeX Live, including `pdflatex`, are available on <https://www.tug.org/texlive/>.

13. Gotran usage

CUDA code generation parameters, along with sample usage, are detailed in Section 8.9.

Parameters specific to the solver interface are detailed in Section 8.10, with sample usage in Section 8.11.

13.1 `gotran2latex`

Terminal output

```
Usage: gotran2latex FILE [options]

Options:
  -h, --help            show this help message and exit
  --auto_format_description=AUTO_FORMAT_DESCRIPTION
                        Default(False): Automatically format
                            state and
                        parameter descriptions
  --bold_equation_labels=BOLD_EQUATION_LABELS
                        Default(True): Give equation labels a
                            bold typeface in
                        LaTeX document
  --columnsep=COLUMNSEP
                        Default(''): Set column separator
                            distance (e.g.
                        '0.25cm'). Uses LaTeX default if left
                            blank
  --columnseprule=COLUMNSEPRULE
                        Default(''): Set column separator line
                            width (e.g.
                        '0.2pt'). Uses LaTeX default if left
                            blank
  --equation_subnumbering=EQUATION_SUBNUMBERING
```

```
                        Default(True): Use component-wise
                            equation
                        subnumbering
--font_size=FONT_SIZE
                        Default(10.0): Set global font size
                            for LaTeX document
--landscape=LANDSCAPE
                        Default(False): Set LaTeX document to
                            landscape layout
--latex_output=LATEX_OUTPUT
                        Default(''): Specify LaTeX output file
--margins=MARGINS       Default(''): Set page margins (e.g.
    '0.75in'). Uses
                        LaTeX defaults if left blank
--math_font_size=MATH_FONT_SIZE
                        Default(0.0): Set font size for
                            mathematical
                        expressions in LaTeX document. Uses
                            global font size
                        if left blank
--mul_symbol=MUL_SYMBOL
                        Default('dot'): Multiplication symbol
                            for Sympy
                        LatexPrinter
--no_page_numbers=NO_PAGE_NUMBERS
                        Default(False): Disable page numbers
--no_parameter_descriptions=NO_PARAMETER_DESCRIPTIONS
                        Default(False): Disable table column
                            for parameter
                        descriptions
--no_preamble=NO_PREAMBLE
                        Default(False): If set to True, LaTeX
                            document will be
                        be generated without the preamble
--no_state_descriptions=NO_STATE_DESCRIPTIONS
                        Default(False): Disable table column
                            for state
                        descriptions
--page_columns=PAGE_COLUMNS
                        Default(1): Set number of columns per
                            page in LaTeX
                        document
--parameter_description_cell_style=PARAMETER_DESCRIPTION_CELL_STYLE
                        Default('l'): Set description cell
                            type for the
```

```
                        parameter table. Use 'X' for long
                            descriptions, or
                        'p{5cm}' to set a fixed 5 cm
  --section_type=SECTION_TYPE
                        Default('section'): Section type (e.g.
                            'section',
                        'subsection')
  --state_description_cell_style=STATE_DESCRIPTION_CELL_STYLE
                        Default('l'): Set description cell
                            type for the state
                        table. Use 'X' for long descriptions,
                            or 'p{5cm}' to
                        set a fixed 5 cm
  --sympy_contraction=SYMPY_CONTRACTION
                        Default(True): If True sympy
                            contraction will be used,
                        turning (V-3)/2 into V/2-3/2
```

14. Additional examples and figures

14.1 Gotran form file for the Ten Tusscher-Panfilov model

The Gotran form file of the Ten Tusscher-Panfilov model [Ten Tusscher and Panfilov, 2006], as provided in Gotran.

Python code

```python
# Alternans and spiral breakup in a human ventricular tissue
    model, K.H.W.J. ten
# Tusscher, A.V. Panfilov, Sep 2006,American Journal of
    Physiology, Heart and
# Circulatory Physiology, 291 3, H1088-1100.PubMed ID: 16565318
#

# gotran file generated by cellml2gotran from
    ten_tusscher_model_2006_IK1Ko_M_units.cellml

parameters("Reversal potentials",
           P_kna = 0.03)

parameters("Inward rectifier potassium current",
           g_K1 = ScalarParam(5.405, unit="nS*pF**-1"))

parameters("Rapid time dependent potassium current",
           g_Kr = ScalarParam(0.153, unit="nS*pF**-1"))
```

```
states("Rapid time dependent potassium current", "Xr1 gate",
       Xr1 = 0.0165)

states("Rapid time dependent potassium current", "Xr2 gate",
       Xr2 = 0.473)

parameters("Slow time dependent potassium current",
           g_Ks = ScalarParam(0.098, unit="nS*pF**-1"))

states("Slow time dependent potassium current", "Xs gate",
       Xs = 0.0174)

parameters("Fast sodium current",
           g_Na = ScalarParam(14.838, unit="nS*pF**-1"))

states("Fast sodium current", "m gate",
       m = 0.00165)

states("Fast sodium current", "h gate",
       h = 0.749)

states("Fast sodium current", "j gate",
       j = 0.6788)

parameters("Sodium background current",
           g_bna = ScalarParam(0.00029, unit="nS*pF**-1"))

parameters("L_type Ca current",
           g_CaL = ScalarParam(0.0000398,
               unit="l*F**-1*s**-1"))

states("L_type Ca current", "d gate",
       d = 3.288e-5)

states("L_type Ca current", "f gate",
       f = 0.7026)

states("L_type Ca current", "F2 gate",
       f2 = 0.9526)

states("L_type Ca current", "FCass gate",
       fCass = 0.9942)

parameters("Calcium background current",
```

```
            g_bca = ScalarParam(0.000592, unit="nS*pF**-1"))

parameters("Transient outward current",
            g_to = ScalarParam(0.294, unit="nS*pF**-1"))

states("Transient outward current", "s gate",
       s = 0.999998)

states("Transient outward current", "r gate",
       r = 2.347e-8)

parameters("Sodium potassium pump current",
            P_NaK = ScalarParam(2.724, unit="pA*pF**-1"),
            K_mk = ScalarParam(1, unit="mM"),
            K_mNa = ScalarParam(40, unit="mM"))

parameters("Sodium calcium exchanger current",
            K_NaCa = ScalarParam(1000, unit="pA*pF**-1"),
            K_sat = 0.1,
            alpha = 2.5,
            gamma = 0.35,
            Km_Ca = ScalarParam(1.38, unit="mM"),
            Km_Nai = ScalarParam(87.5, unit="mM"))

parameters("Calcium pump current",
            g_pCa = ScalarParam(0.1238, unit="pA*pF**-1"),
            K_pCa = ScalarParam(0.0005, unit="mM"))

parameters("Potassium pump current",
            g_pK = ScalarParam(0.0146, unit="nS*pF**-1"))

states("Calcium dynamics",
       R_prime = 0.8978,
       Ca_i = ScalarParam(0.000153, unit="mM"),
       Ca_SR = ScalarParam(4.272, unit="mM"),
       Ca_ss = ScalarParam(0.00042, unit="mM"))

parameters("Calcium dynamics",
            Ca_o = ScalarParam(2, unit="mM"),
            k1_prime = ScalarParam(0.15, unit="mM**-2*ms**-1"),
            k2_prime = ScalarParam(0.045, unit="mM**-1*ms**-1"),
            k3 = ScalarParam(0.06, unit="ms**-1"),
            k4 = ScalarParam(0.005, unit="ms**-1"),
            EC = ScalarParam(1.5, unit="mM"),
            max_sr = 2.5,
```

```
            min_sr = 1,
            V_rel = ScalarParam(0.102, unit="ms**-1"),
            V_xfer = ScalarParam(0.0038, unit="ms**-1"),
            K_up = ScalarParam(0.00025, unit="mM"),
            V_leak = ScalarParam(0.00036, unit="ms**-1"),
            Vmax_up = ScalarParam(0.006375, unit="mM*ms**-1"),
            Buf_c = ScalarParam(0.2, unit="mM"),
            K_buf_c = ScalarParam(0.001, unit="mM"),
            Buf_sr = ScalarParam(10, unit="mM"),
            K_buf_sr = ScalarParam(0.3, unit="mM"),
            Buf_ss = ScalarParam(0.4, unit="mM"),
            K_buf_ss = ScalarParam(0.00025, unit="mM"),
            V_sr = ScalarParam(0.001094, unit="um**3"),
            V_ss = ScalarParam(0.00005468, unit="um**3"))

states("Sodium dynamics",
       Na_i = ScalarParam(10.132, unit="mM"))

parameters("Sodium dynamics",
            Na_o = ScalarParam(140, unit="mM"))

states("Membrane",
       V = ScalarParam(-85.423, unit="mV"))

parameters("Membrane",
            R = ScalarParam(8314.472, unit="J*mole**-1*K**-1"),
            T = ScalarParam(310, unit="K"),
            F = ScalarParam(96485.3415, unit="C*mmole**-1"),
            Cm = ScalarParam(0.185, unit="uF"),
            V_c = ScalarParam(0.016404, unit="um**3"),
            stim_start = ScalarParam(10, unit="ms"),
            stim_period = ScalarParam(1000, unit="ms"),
            stim_duration = ScalarParam(1, unit="ms"),
            stim_amplitude = ScalarParam(52, unit="pA*pF**-1"))

states("Potassium dynamics",
       K_i = ScalarParam(138.52, unit="mM"))

parameters("Potassium dynamics",
            K_o = ScalarParam(5.4, unit="mM"))

component("Reversal potentials")
E_Na = R*T/F*log(Na_o/Na_i) # mV
E_K = R*T/F*log(K_o/K_i) # mV
E_Ks = R*T/F*log((K_o + P_kna*Na_o)/(K_i + P_kna*Na_i)) # mV
```

```
E_Ca = 0.5*R*T/F*log(Ca_o/Ca_i) # mV

component("Inward rectifier potassium current")
alpha_K1 = 0.1/(1 + exp(0.06*(V - E_K - 200)))
beta_K1 = (3*exp(0.0002*(V - E_K + 100)) + exp(0.1*(V - E_K -
    10)))/(1 + exp(-0.5*(V - E_K)))
xK1_inf = alpha_K1/(alpha_K1 + beta_K1)
i_K1 = g_K1*xK1_inf*sqrt(K_o/5.4)*(V - E_K) # pA*pF**-1

component("Rapid time dependent potassium current")
i_Kr = g_Kr*sqrt(K_o/5.4)*Xr1*Xr2*(V - E_K) # pA*pF**-1

component("Rapid time dependent potassium current", "Xr1 gate")
xr1_inf = 1/(1 + exp((-26 - V)/7))
alpha_xr1 = 450/(1 + exp((-45 - V)/10))
beta_xr1 = 6/(1 + exp((V + 30)/11.5))
tau_xr1 = 1*alpha_xr1*beta_xr1 # ms
dXr1_dt = (xr1_inf - Xr1)/tau_xr1

component("Rapid time dependent potassium current", "Xr2 gate")
xr2_inf = 1/(1 + exp((V + 88)/24))
alpha_xr2 = 3/(1 + exp((-60 - V)/20))
beta_xr2 = 1.12/(1 + exp((V - 60)/20))
tau_xr2 = 1*alpha_xr2*beta_xr2 # ms
dXr2_dt = (xr2_inf - Xr2)/tau_xr2

component("Slow time dependent potassium current")
i_Ks = g_Ks*Xs**2*(V - E_Ks) # pA*pF**-1

component("Slow time dependent potassium current", "Xs gate")
xs_inf = 1/(1 + exp((-5 - V)/14))
alpha_xs = 1400/sqrt(1 + exp((5 - V)/6))
beta_xs = 1/(1 + exp((V - 35)/15))
tau_xs = 1*alpha_xs*beta_xs + 80 # ms
dXs_dt = (xs_inf - Xs)/tau_xs

component("Fast sodium current")
i_Na = g_Na*m**3*h*j*(V - E_Na) # pA*pF**-1

component("Fast sodium current", "m gate")
m_inf = 1/(1 + exp((-56.86 - V)/9.03))**2
alpha_m = 1/(1 + exp((-60 - V)/5))
beta_m = 0.1/(1 + exp((V + 35)/5)) + 0.1/(1 + exp((V -
    50)/200))
tau_m = 1*alpha_m*beta_m # ms
```

```
dm_dt = (m_inf - m)/tau_m

component("Fast sodium current", "h gate")
h_inf = 1/(1 + exp((V + 71.55)/7.43))**2
alpha_h = Conditional(Lt(V, -40), 0.057*exp(-(V + 80)/6.8), 0)
    # ms**-1
beta_h = Conditional(Lt(V, -40), 2.7*exp(0.079*V) +
    310000*exp(0.3485*V), 0.77/(0.13*(1 + exp((V +
    10.66)/-11.1)))) # ms**-1
tau_h = 1/(alpha_h + beta_h) # ms
dh_dt = (h_inf - h)/tau_h

component("Fast sodium current", "j gate")
j_inf = 1/(1 + exp((V + 71.55)/7.43))**2
alpha_j = Conditional(Lt(V, -40), (-25428*exp(0.2444*V) -
    6.948e-6*exp(-0.04391*V))*(V + 37.78)/1/(1 + exp(0.311*(V +
     79.23))), 0) # ms**-1
beta_j = Conditional(Lt(V, -40), 0.02424*exp(-0.01052*V)/(1 +
    exp(-0.1378*(V + 40.14))), 0.6*exp(0.057*V)/(1 +
    exp(-0.1*(V + 32)))) # ms**-1
tau_j = 1/(alpha_j + beta_j) # ms
dj_dt = (j_inf - j)/tau_j

component("Sodium background current")
i_b_Na = g_bna*(V - E_Na) # pA*pF**-1

component("L_type Ca current")
V_eff = Conditional(Lt(abs(V-15), 1.e-2), 1e-2, V-15)
i_CaL = g_CaL*d*f*f2*fCass*4*V_eff*F**2/(R*T) *
    (0.25*Ca_ss*exp(2*V_eff*F/(R*T)) -
    Ca_o)/(exp(2*V_eff*F/(R*T)) - 1) # pA*pF**-1

component("L_type Ca current", "d gate")
d_inf = 1/(1 + exp((-8 - V)/7.5))
alpha_d = 1.4/(1 + exp((-35 - V)/13)) + 0.25
beta_d = 1.4/(1 + exp((V + 5)/5))
gamma_d = 1/(1 + exp((50 - V)/20)) # ms
tau_d = 1*alpha_d*beta_d + gamma_d # ms
dd_dt = (d_inf - d)/tau_d

component("L_type Ca current", "f gate")
f_inf = 1/(1 + exp((V + 20)/7))
tau_f = 1102.5*exp(-((V + 27)**2)/225) + 200/(1 + exp((13 -
    V)/10)) + 180/(1 + exp((V + 30)/10)) + 20 # ms
df_dt = (f_inf - f)/tau_f
```

```
component("L_type Ca current", "F2 gate")
f2_inf = 0.67/(1 + exp((V + 35)/7)) + 0.33
tau_f2 = 562*exp(-((V + 27)**2)/240) + 31/(1 + exp((25 -
    V)/10)) + 80/(1 + exp((V + 30)/10)) # ms
df2_dt = (f2_inf - f2)/tau_f2

component("L_type Ca current", "FCass gate")
fCass_inf = 0.6/(1 + (Ca_ss/0.05)**2) + 0.4
tau_fCass = 80/(1 + (Ca_ss/0.05)**2) + 2 # ms
dfCass_dt = (fCass_inf - fCass)/tau_fCass

component("Calcium background current")
i_b_Ca = g_bca*(V - E_Ca) # pA*pF**-1

component("Transient outward current")
i_to = g_to*r*s*(V - E_K) # pA*pF**-1

component("Transient outward current", "s gate")
s_inf = 1/(1 + exp((V + 20)/5))
tau_s = 85*exp(-((V + 45)**2)/320) + 5/(1 + exp((V - 20)/5)) +
    3 # ms
ds_dt = (s_inf - s)/tau_s

component("Transient outward current", "r gate")
r_inf = 1/(1 + exp((20 - V)/6))
tau_r = 9.5*exp(-((V + 40)**2)/1800) + 0.8 # ms
dr_dt = (r_inf - r)/tau_r

component("Sodium potassium pump current")
i_NaK = P_NaK*K_o/(K_o + K_mk)*Na_i/(Na_i + K_mNa)/(1 +
    0.1245*exp(-0.1*V*F/(R*T)) + 0.0353*exp(-V*F/(R*T))) #
    pA*pF**-1

component("Sodium calcium exchanger current")
i_NaCa = K_NaCa*(exp(gamma*V*F/(R*T))*Na_i**3*Ca_o -
    exp((gamma - 1)*V*F/(R*T))*Na_o**3*Ca_i*alpha)/((Km_Nai**3
    + Na_o**3)*(Km_Ca + Ca_o)*(1 + K_sat*exp((gamma -
    1)*V*F/(R*T)))) # pA*pF**-1

component("Calcium pump current")
i_p_Ca = g_pCa*Ca_i/(Ca_i + K_pCa) # pA*pF**-1

component("Potassium pump current")
i_p_K = g_pK*(V - E_K)/(1 + exp((25 - V)/5.98)) # pA*pF**-1
```

```
component("Calcium dynamics")
i_up = Vmax_up/(1 + K_up**2/Ca_i**2) # mM*ms**-1
i_leak = V_leak*(Ca_SR - Ca_i) # mM*ms**-1
i_xfer = V_xfer*(Ca_ss - Ca_i) # mM*ms**-1
kcasr = max_sr - (max_sr - min_sr)/(1 + (EC/Ca_SR)**2)
Ca_i_bufc = 1/(1 + Buf_c*K_buf_c/(Ca_i + K_buf_c)**2)
Ca_sr_bufsr = 1/(1 + Buf_sr*K_buf_sr/(Ca_SR + K_buf_sr)**2)
Ca_ss_bufss = 1/(1 + Buf_ss*K_buf_ss/(Ca_ss + K_buf_ss)**2)
dCa_i_dt = Ca_i_bufc*((i_leak - i_up)*V_sr/V_c + i_xfer -
    1*(i_b_Ca + i_p_Ca - 2*i_NaCa)*Cm/(2*1*V_c*F))
k1 = k1_prime/kcasr # mM**-2*ms**-1
k2 = k2_prime*kcasr # mM**-1*ms**-1
O = k1*Ca_ss**2*R_prime/(k3 + k1*Ca_ss**2)
dR_prime_dt = -k2*Ca_ss*R_prime + k4*(1 - R_prime)
i_rel = V_rel*O*(Ca_SR - Ca_ss) # mM*ms**-1
dCa_SR_dt = Ca_sr_bufsr*(i_up - (i_rel + i_leak))
dCa_ss_dt = Ca_ss_bufss*(-1*i_CaL*Cm/(2*1*V_ss*F) +
    i_rel*V_sr/V_ss - i_xfer*V_c/V_ss)

component("Sodium dynamics")
dNa_i_dt = -1*(i_Na + i_b_Na + 3*i_NaK + 3*i_NaCa)/(1*V_c*F)*Cm

component("Membrane")
i_Stim = Conditional(And(Ge(time -
    floor(time/stim_period)*stim_period, stim_start), Le(time -
     floor(time/stim_period)*stim_period, stim_start +
    stim_duration), ), -stim_amplitude, 0) # pA*pF**-1
dV_dt = -(i_K1 + i_to + i_Kr + i_Ks + i_CaL + i_NaK + i_Na +
    i_b_Na + i_NaCa + i_b_Ca + i_p_K + i_p_Ca + i_Stim)

component("Potassium dynamics")
dK_i_dt = -1*(i_K1 + i_to + i_Kr + i_Ks + i_p_K + i_Stim -
    2*i_NaK)/(1*V_c*F)*Cm
```

14.2 Generated PDF file of the Ten Tusscher-Panfilov model

Figure 14.1 is an embedded PDF file of the automatically generated LaTeX code of the Ten Tusscher-Panfilov model [Ten Tusscher and Panfilov, 2006] using the Gotran form file in Section 14.1. The LaTeX code was generated with:

Bash code

```
gotran2latex tentusscher_panfilov_2006_M_cell.ode
    --font_size=9 --math_font_size=7.5 --page_columns=2
    --latex_output=tentusscher_panfilov.tex
    --no_page_numbers=1 --section_type=subsection
    --columnsep=0.25cm --columnseprule=0.2pt --margins=0.75in
    --auto_format_description=1
```

A small font size has been used to conserve space.

Equations are broken up into separate lines automatically. For some equations, this automatic process fails. In this example, there were three instances of equations bleeding over their margins. These equations have been split manually by introducing additional intermediate variables $c_{\beta_{K1}}$, c_{α_j} and $T_{i_{Stim}}$. These manual refactorisations are shown in red.

Parameters

Table 1: **Parameter Table**

Parameter	Value	Description
P_{kna}	30×10^{-3}	Permeability of Na
g_{K1}	5.41 nS pF^{-1}	I_{K1} base conductivity
g_{Kr}	0.15 nS pF^{-1}	I_{Kr} base conductivity
g_{Ks}	98×10^{-3} nS pF^{-1}	I_{Ks} base conductivity
g_{Na}	14.8 nS pF^{-1}	I_{Na} base conductivity
g_{bna}	290×10^{-6} nS pF^{-1}	$I_{b_{Na}}$ base conductivity
g_{CaL}	39.8×10^{-6} l F^{-1} s^{-1}	I_{CaL} base conductivity
g_{bca}	592×10^{-6} nS pF^{-1}	$I_{b_{Ca}}$ base conductivity
g_{to}	0.29 nS pF^{-1}	I_{to} base conductivity
K_{mNa}	40 mM	Na dissociation constant for I_{NaK}
K_{mk}	1 mM	K dissociation constant for I_{NaK}
P_{NaK}	2.72 pA pF^{-1}	I_{NaK} base rate
K_{NaCa}	1×10^{3} pA pF^{-1}	I_{NaCa} base rate
K_{sat}	0.1	
Km_{Ca}	1.38 mM	Ca_o dissociation constant for I_{NaCa}
Km_{Nai}	87.5 mM	Na_i dissociation constant for I_{NaCa}
α	2.5	
γ	0.35	
K_{pCa}	500×10^{-6} mM	Ca_i dissociation constant for I_{pCa}
g_{pCa}	0.12 pA pF^{-1}	I_{pCa} base conductivity
g_{pK}	14.6×10^{-3} nS pF^{-1}	I_{pK} base conductivity
Buf_c	0.2 mM	Total Ca buffer capacity in Cytosole
Buf_{sr}	10 mM	Total Ca buffer capacity in SR
Buf_{ss}	0.4 mM	Total Ca buffer capacity in sub space
Ca_o	2 mM	External Ca
EC	1.5 mM	RyR SR Ca scale value
K_{bufc}	1×10^{-3} mM	Ca dissociation constant for buffer in Cytosole
K_{bufsr}	0.3 mM	Ca dissociation constant for buffer in SR
K_{bufss}	250×10^{-6} mM	Ca dissociation constant for buffer in sub space
K_{up}	250×10^{-6} mM	Ca dissociation constant for SERCA pump
V_{leak}	360×10^{-6} ms^{-1}	Ca leak rate
V_{rel}	0.10 ms^{-1}	RyR base release rate
V_{sr}	1.09×10^{-3} μm^3	Volume SR
V_{ss}	54.7×10^{-6} μm^3	Volume sub space
V_{xfer}	3.8×10^{-3} ms^{-1}	Ca base transfer rate
$Vmax_{up}$	6.37×10^{-3} mM ms^{-1}	Ca base rate SERCA pump
k_{1prime}	0.15 mM^{-2} ms^{-1}	RyR opening rate
k_{2prime}	45×10^{-3} mM^{-1} ms^{-1}	RyR inactivation rate
k_3	60×10^{-3} ms^{-1}	RyR deactivation rate
k_4	5×10^{-3} ms^{-1}	RyR return from inactivation rate
max_{sr}	2.5 mM	RyR max SR Ca scale value
min_{sr}	1 mM	RyR min SR Ca scale value
Na_o	140 mM	Extracellular Na
Cm	0.18 μF	Faraday' s constant
F	96.5×10^{3} C mmole^{-1}	
R	8.31×10^{3} J mole^{-1} K^{-1}	Universal gass constant
T	310 K	Temperature
V_c	16.4×10^{-3} μm^3	Volume cytosole
$stim_{amplitude}$	52 pA pF^{-1}	Amplitude for stimulation
$stim_{duration}$	1 ms	Duration time for stimulation
$stim_{period}$	1×10^{3} ms	Timer period for stimulation
$stim_{start}$	10 ms	Start time for stimulation
K_o	5.4 mM	Extracellular K

Figure 14.1: *Ten Tusscher-Panfilov model, p. 1*

Initial Values

Table 2: **State Table**

State	Value	Description
Xr_1	16.5×10^{-3}	$Xr1$ gate in I_{Kr}
Xr_2	0.47	$Xr2$ gate in I_{Kr}
Xs	17.4×10^{-3}	Xs gate in I_{Ks}
m	1.65×10^{-3}	m gate in I_{Na}
h	0.75	h gate in I_{Na}
j	0.68	j gate in I_{Na}
d	32.9×10^{-6}	d gate in I_{CaL}
f	0.70	f gate in I_{CaL}
f_2	0.95	$f2$ gate in I_{CaL}
$fCass$	0.99	FCass gate in I_{CaL}
s	1	s gate in I_{to}
r	23.5×10^{-9}	s gate in I_{to}
Ca_i	153×10^{-6} mM	Intracellular Ca
R_{prime}	0.90	RyR availability
Ca_{SR}	4.27 mM	SR Ca
Ca_{ss}	420×10^{-6} mM	Subspace Ca
Na_i	10.1 mM	Intracellular Na
V	−85.4 mV	Membrane potential
K_i	139 mM	Intracellular K

Components

Reversal potentials

$$E_{Na} = \frac{RT}{F}\log\left(\frac{Na_o}{Na_i}\right) \tag{1a}$$

$$E_K = \frac{RT}{F}\log\left(\frac{K_o}{K_i}\right) \tag{1b}$$

$$E_{Ks} = \frac{RT}{F}\log\left(\frac{Na_o P_{kna} + K_o}{K_i + P_{kna} Na_i}\right) \tag{1c}$$

$$E_{Ca} = \frac{0.5R}{F}T\log\left(\frac{Ca_o}{Ca_i}\right) \tag{1d}$$

Inward rectifier potassium current

$$\alpha_{K1} = \frac{0.1}{1 + 6.14\times10^{-6} e^{60\times10^{-3}V - 60\times10^{-3}E_K}} \tag{2a}$$

$$c_{\beta_{K1}} = \left(0.37e^{-0.1E_K+0.1V} + 3.06e^{-200\times10^{-6}E_K+200\times10^{-6}V}\right) \tag{2b}$$

$$\beta_{K1} = \frac{1}{1 + e^{-0.5V+0.5E_K}} c_{\beta_{K1}} \tag{2c}$$

$$xK1_{inf} = \frac{\alpha_{K1}}{\beta_{K1} + \alpha_{K1}} \tag{2d}$$

$$i_{K1} = 0.43 g_{K1}\sqrt{K_o}\,(-E_K + V)\,xK1_{inf} \tag{2e}$$

Rapid time dependent potassium current

$$i_{Kr} = 0.43 g_{Kr}\sqrt{K_o}\,(-E_K + V)\,Xr_1 Xr_2 \tag{3a}$$

Xr1 gate

$$xr_{1inf} = \frac{1}{1 + e^{-\frac{26}{7} - \frac{V}{7}}} \tag{4a}$$

$$\alpha_{xr1} = \frac{450}{1 + e^{-\frac{9}{2} - \frac{V}{10}}} \tag{4b}$$

$$\beta_{xr1} = \frac{6}{1 + 13.6e^{87\times10^{-3}V}} \tag{4c}$$

$$\tau_{xr1} = \alpha_{xr1}\beta_{xr1} \tag{4d}$$

$$\frac{dXr_1}{dt} = \frac{1}{\tau_{xr1}}\left(-Xr_1 + xr_{1inf}\right) \tag{4e}$$

Xr2 gate

$$xr_{2inf} = \frac{1}{1 + e^{\frac{11}{3} + \frac{V}{24}}} \tag{5a}$$

$$\alpha_{xr2} = \frac{3}{1 + e^{-3 - \frac{V}{20}}} \tag{5b}$$

$$\beta_{xr2} = \frac{1.12}{1 + e^{-3 + \frac{V}{20}}} \tag{5c}$$

$$\tau_{xr2} = \alpha_{xr2}\beta_{xr2} \tag{5d}$$

$$\frac{dXr_2}{dt} = \frac{1}{\tau_{xr2}}\left(-Xr_2 + xr_{2inf}\right) \tag{5e}$$

Figure 14.1: *Ten Tusscher-Panfilov model, p. 2*

Slow time dependent potassium current

$$i_{Ks} = g_{Ks} Xs^2 \left(-E_{Ks} + V\right) \tag{6a}$$

Xs gate

$$xs_{inf} = \frac{1}{1 + e^{-\frac{5}{14} - \frac{V}{14}}} \tag{7a}$$

$$\alpha_{xs} = \frac{1.4 \times 10^3}{\sqrt{1 + e^{\frac{5}{6} - \frac{V}{6}}}} \tag{7b}$$

$$\beta_{xs} = \frac{1}{1 + e^{-\frac{7}{3} + \frac{V}{15}}} \tag{7c}$$

$$\tau_{xs} = 80 + \alpha_{xs} \beta_{xs} \tag{7d}$$

$$\frac{dXs}{dt} = \frac{1}{\tau_{xs}} \left(xs_{inf} - Xs\right) \tag{7e}$$

Fast sodium current

$$i_{Na} = g_{Na} m^3 \left(V - E_{Na}\right) hj \tag{8a}$$

m gate

$$m_{inf} = \frac{1}{\left(1 + 1.84 \times 10^{-3} e^{-0.11V}\right)^2} \tag{9a}$$

$$\alpha_m = \frac{1}{1 + e^{-12 - \frac{V}{5}}} \tag{9b}$$

$$\beta_m = \frac{0.1}{1 + e^{-\frac{1}{4} + \frac{V}{200}}} + \frac{0.1}{1 + e^{7 + \frac{V}{5}}} \tag{9c}$$

$$\tau_m = \alpha_m \beta_m \tag{9d}$$

$$\frac{dm}{dt} = \frac{1}{\tau_m} \left(-m + m_{inf}\right) \tag{9e}$$

h gate

$$h_{inf} = \frac{1}{\left(1 + 15.2 \times 10^3 e^{0.14V}\right)^2} \tag{10a}$$

$$\alpha_h = \begin{cases} 443 \times 10^{-9} e^{-0.15V} & \text{for } V < -40 \\ 0 & \text{otherwise} \end{cases} \tag{10b}$$

$$\beta_h = \begin{cases} 2.7 e^{79 \times 10^{-3} V} + 310 \times 10^3 e^{0.35V} & \text{for } V < -40 \\ \frac{0.77}{0.13 + 49.8 \times 10^{-3} e^{-90.1 \times 10^{-3} V}} & \text{otherwise} \end{cases} \tag{10c}$$

$$\tau_h = \frac{1}{\beta_h + \alpha_h} \tag{10d}$$

$$\frac{dh}{dt} = \frac{1}{\tau_h} \left(-h + h_{inf}\right) \tag{10e}$$

j gate

$$j_{inf} = \frac{1}{\left(1 + 15.2 \times 10^3 e^{0.14V}\right)^2} \tag{11a}$$

$$c_{\alpha_j} = \left(-25.4 \times 10^3 e^{0.24V} - 6.95 \times 10^{-6} e^{-43.9 \times 10^{-3} V}\right) \tag{11b}$$

$$\alpha_j = \begin{cases} \frac{1}{1 + 50.3 \times 10^9 e^{0.31V}} \left(37.8 + V\right) c_{\alpha_j} & \text{for } V < -40 \\ 0 & \text{otherwise} \end{cases} \tag{11c}$$

$$\beta_j = \begin{cases} \frac{24.2 \times 10^{-3} e^{-10.5 \times 10^{-3} V}}{1 + 3.96 \times 10^{-3} e^{-0.14V}} & \text{for } V < -40 \\ \frac{0.6 e^{57 \times 10^{-3} V}}{1 + 40.8 \times 10^{-3} e^{-0.1V}} & \text{otherwise} \end{cases} \tag{11d}$$

$$\tau_j = \frac{1}{\beta_j + \alpha_j} \tag{11e}$$

$$\frac{dj}{dt} = \frac{1}{\tau_j} \left(-j + j_{inf}\right) \tag{11f}$$

Sodium background current

$$i_{bNa} = g_{bna} \left(V - E_{Na}\right) \tag{12a}$$

L_type Ca current

$$V_{eff} = \begin{cases} 10 \times 10^{-3} & \text{for } |-15 + V| < 10 \times 10^{-3} \\ -15 + V & \text{otherwise} \end{cases} \tag{13a}$$

$$i_{CaL} = \frac{4 g_{CaL} F^2 \left(0.25 Ca_{ss} e^{\frac{2FV_{eff}}{RT}} - Ca_o\right) V_{eff} df f_2 f Cass}{RT \left(-1 + e^{\frac{2FV_{eff}}{RT}}\right)} \tag{13b}$$

d gate

$$d_{inf} = \frac{1}{1 + 0.34 e^{-0.13V}} \tag{14a}$$

$$\alpha_d = 0.25 + \frac{1.4}{1 + e^{-\frac{35}{13} - \frac{V}{13}}} \tag{14b}$$

$$\beta_d = \frac{1.4}{1 + e^{1 + \frac{V}{5}}} \tag{14c}$$

$$\gamma_d = \frac{1}{1 + e^{\frac{5}{2} - \frac{V}{20}}} \tag{14d}$$

$$\tau_d = \gamma_d + \alpha_d \beta_d \tag{14e}$$

$$\frac{dd}{dt} = \frac{1}{\tau_d} \left(-d + d_{inf}\right) \tag{14f}$$

f gate

$$f_{inf} = \frac{1}{1 + e^{\frac{20}{7} + \frac{V}{7}}} \tag{15a}$$

$$\tau_f = 20 + 1.1 \times 10^3 e^{-\frac{1}{225}(27+V)^2} + \frac{180}{1 + e^{3 + \frac{V}{10}}} + \frac{200}{1 + e^{\frac{13}{10} - \frac{V}{10}}} \tag{15b}$$

$$\frac{df}{dt} = \frac{1}{\tau_f} \left(-f + f_{inf}\right) \tag{15c}$$

Figure 14.1: *Ten Tusscher-Panfilov model, p. 3*

F2 gate

$$f_{2inf} = 0.33 + \frac{0.67}{1 + e^{5 + \frac{V}{7}}} \tag{16a}$$

$$\tau_{f2} = \frac{31}{1 + e^{\frac{5}{2} - \frac{V}{10}}} + 562 e^{-\frac{1}{240}(27+V)^2} + \frac{80}{1 + e^{3 + \frac{V}{10}}} \tag{16b}$$

$$\frac{df_2}{dt} = \frac{1}{\tau_{f2}} \left(-f_2 + f_{2inf} \right) \tag{16c}$$

FCass gate

$$fCass_{inf} = 0.4 + \frac{0.6}{1 + 400 Ca_{ss}^2} \tag{17a}$$

$$\tau_{fCass} = 2 + \frac{80}{1 + 400 Ca_{ss}^2} \tag{17b}$$

$$\frac{dfCass}{dt} = \frac{1}{\tau_{fCass}} \left(-fCass + fCass_{inf} \right) \tag{17c}$$

Calcium background current

$$i_{bCa} = g_{bca} \left(-E_{Ca} + V \right) \tag{18a}$$

Transient outward current

$$i_{to} = g_{to} \left(-E_K + V \right) rs \tag{19a}$$

s gate

$$s_{inf} = \frac{1}{1 + e^{4 + \frac{V}{5}}} \tag{20a}$$

$$\tau_s = 3 + 85 e^{-\frac{1}{320}(45+V)^2} + \frac{5}{1 + e^{-4 + \frac{V}{5}}} \tag{20b}$$

$$\frac{ds}{dt} = \frac{1}{\tau_s} \left(s_{inf} - s \right) \tag{20c}$$

r gate

$$r_{inf} = \frac{1}{1 + e^{\frac{10}{3} - \frac{V}{6}}} \tag{21a}$$

$$\tau_r = 0.8 + 9.5 e^{-\frac{(40+V)^2}{1.8 \times 10^3}} \tag{21b}$$

$$\frac{dr}{dt} = \frac{1}{\tau_r} \left(-r + r_{inf} \right) \tag{21c}$$

Sodium potassium pump current

$$i_{NaK} = \frac{K_o P_{NaK} Na_i}{\left(K_{mNa} + Na_i \right) \left(K_{mk} + K_o \right) \left(1 + 0.12 e^{-\frac{0.1FV}{RT}} + 35.3 \times 10^{-3} e^{-\frac{FV}{RT}} \right)} \tag{22a}$$

Sodium calcium exchanger current

$$i_{NaCa} = \frac{K_{NaCa} \left(-\alpha Na_o^3 Ca_i e^{\frac{FV}{RT}(-1+\gamma)} + Ca_o Na_i^3 e^{\frac{F\gamma V}{RT}} \right)}{\left(1 + K_{sat} e^{\frac{FV}{RT}(-1+\gamma)} \right) \left(Ca_o + Km_{Ca} \right) \left(Km_{Nai}^3 + Na_o^3 \right)} \tag{23a}$$

Calcium pump current

$$i_{pCa} = \frac{g_{pCa} Ca_i}{Ca_i + K_{pCa}} \tag{24a}$$

Potassium pump current

$$i_{pK} = \frac{g_{pK} \left(-E_K + V \right)}{1 + 65.4 e^{-0.17V}} \tag{25a}$$

Calcium dynamics

$$i_{up} = \frac{Vmax_{up}}{1 + \frac{K_{up}^2}{Ca_i^2}} \tag{26a}$$

$$i_{leak} = V_{leak} \left(Ca_{SR} - Ca_i \right) \tag{26b}$$

$$i_{xfer} = V_{xfer} \left(Ca_{ss} - Ca_i \right) \tag{26c}$$

$$kcasr = max_{sr} - \frac{max_{sr} - min_{sr}}{1 + \frac{EC^2}{Ca_{SR}^2}} \tag{26d}$$

$$Ca_{ibufc} = \frac{1}{1 + \frac{Buf_c K_{bufc}}{\left(Ca_i + K_{bufc} \right)^2}} \tag{26e}$$

$$Ca_{srbufsr} = \frac{1}{1 + \frac{Buf_{sr} K_{bufsr}}{\left(Ca_{SR} + K_{bufsr} \right)^2}} \tag{26f}$$

$$Ca_{ssbufss} = \frac{1}{1 + \frac{Buf_{ss} K_{bufss}}{\left(Ca_{ss} + K_{bufss} \right)^2}} \tag{26g}$$

$$\frac{dCa_i}{dt} = \left(\frac{V_{sr}}{V_c} \left(-i_{up} + i_{leak} \right) + i_{xfer} - \frac{Cm}{2FV_c} \left(i_{bCa} - 2i_{NaCa} + i_{pCa} \right) \right) Ca_{ibufc} \tag{26h}$$

$$k_1 = \frac{k_{1prime}}{kcasr} \tag{26i}$$

$$k_2 = k_{2prime} kcasr \tag{26j}$$

$$O = \frac{Ca_{ss}^2 R_{prime} k_1}{k_3 + Ca_{ss}^2 k_1} \tag{26k}$$

$$\frac{dR_{prime}}{dt} = -Ca_{ss} R_{prime} k_2 + k_4 \left(1 - R_{prime} \right) \tag{26l}$$

$$i_{rel} = V_{rel} \left(Ca_{SR} - Ca_{ss} \right) O \tag{26m}$$

$$\frac{dCa_{SR}}{dt} = \left(-i_{rel} - i_{leak} + i_{up} \right) Ca_{srbufsr} \tag{26n}$$

Figure 14.1: *Ten Tusscher-Panfilov model, p. 4*

$$\frac{dCa_{ss}}{dt} = \left(-\frac{Cmi_{CaL}}{2FV_{ss}} - \frac{V_c i_{xfer}}{V_{ss}} + \frac{V_{sr} i_{rel}}{V_{ss}}\right) Ca_{ssbuf_{ss}} \quad (26o)$$

Sodium dynamics

$$\frac{dNa_i}{dt} = \frac{Cm}{FV_c}\left(-3i_{NaCa} - i_{bNa} - i_{Na} - 3i_{NaK}\right) \quad (27a)$$

Membrane

$$T_{i_{Stim}} = stim_{start} \wedge -stim_{period}\lfloor\frac{t}{stim_{period}}\rfloor + t \leq stim_{start} + stim_{duration} \quad (28a)$$

$$i_{Stim} = \begin{cases} -stim_{amplitude} & \text{for } -stim_{period}\lfloor\frac{t}{stim_{period}}\rfloor + t \geq T_{i_{Stim}} \\ 0 & \text{otherwise} \end{cases} \quad (28b)$$

$$\frac{dV}{dt} = -i_{K1} - i_{Ks} - i_{bCa} - i_{pCa} - i_{Kr} - i_{pK} - i_{Stim} - i_{CaL} - i_{NaK} - i_{bNa} - i_{NaCa} - i_{Na} - i_{to} \quad (28c)$$

Potassium dynamics

$$\frac{dK_i}{dt} = \frac{Cm}{FV_c}\left(-i_{K1} - i_{Ks} + 2i_{NaK} - i_{Kr} - i_{pK} - i_{Stim} - i_{to}\right) \quad (29a)$$

Figure 14.1: *Ten Tusscher-Panfilov model, p. 5*

List of Figures

List of Tables

Bibliography

Ackermann J, Baecher P, Franzel T, Goesele M, Hamacher K et al. (2009). Massively-parallel simulation of biochemical systems. In *GI Jahrestagung*, pp. 739–750. Citeseer. 8.1

Ahnert K & Mulansky M (2011). Odeint - solving ordinary differential equations in c++ <http://headmyshoulder.github.com/odeint-v2>. 5.1

Beeler GW & Reuter H (1977). Reconstruction of the action potential of ventricular myocardial fibres. *J Physiol* **268**, 177–210. 2.2, 2.3

Butcher JC (1987). *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods* Wiley-Interscience. 3.1.2

Carlisle D (2004). *The longtable package* <http://ctan.uib.no/macros/latex/required/tools/longtable.pdf>. 7.2.1

Chervet F (2011). *tabu and longtabu – Flexible LaTeX tabulars* <http://ctan.uib.no/macros/latex/contrib/tabu/tabu.pdf>. 7.2.1

Cook S (2013). *CUDA programming: a developer's guide to parallel computing with GPUs* Newnes. 4.1.3

Flaim SN & McCulloch AD (2007). Acetylcholine-induced shortening of the epicardial action potential duration may increase repolarization gradients and lqt3 arrhythmic risk. *J Electrocardiol* **40**, S66–S69. 2.2

Grandi E, Pasqualini FS & Bers DM (2010). A novel computational model of the human ventricular action potential and Ca^{2+} transient. *J Mol Cell Cardiol* **48**, 112–121. 2.2, 2.3

Hairer E & Wanner G (1999). Stiff differential equations solved by radau methods. *Journal of Computational and Applied Mathematics* **111**, 93–111. 3.2

Hindmarsh A, Brown P, Grant K, Lee S, Serban R, Shumaker D & Woodward C (2005). Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)* **31**, 363–396. 5.1

Hodgkin AL & Huxley AF (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology* **117**, 500. 2.1, 2.3

Høgholm M, Downes MJ, Madsen L, Robertson W & Wright J (2012). *The breqn package* <http://ctan.uib.no/macros/latex/contrib/mh/breqn.pdf> Accessed: 2014-03-28. 7.3.1

Kaarby M (2007). Kombinasjonen av eksplisitt og implisitt løser for simulering av den elektriske aktiviteten i hjertet Master's thesis, Norwegian University of Science and Technology. 1

Klöckner A (2014). Pycuda 2013.1.1 documentation <http://documen.tician.de/pycuda/> Accessed: 2014-05-29. 4.1, 4.1.4, 8.2

Langtangen HP (2008). *Python Scripting for Computational Science* Springer, 3rd edition. 5.2

Lionetti F (2010). Gpu accelerated cardiac electrophysiology Master's thesis, University of California, San Diego. 3.1.4, 10.1

Luebke D, Harris M, Govindaraju N, Lefohn A, Houston M, Owens J, Segal M, Papakipos M & Buck I (2006). Gpgpu: general-purpose computation on graphics hardware In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 208. ACM. 4

Mardal KA, Skavhaug O, Lines GT, Staff GA & Ødegård A (2007). Using python to solve partial differential equations. *Computing in Science & Engineering* **9**, 48–51. 11

McAllister RE, Noble D & Tsien R (1975). Reconstruction of the electrical activity of cardiac purkinje fibres. *The Journal of physiology* **251**, 1–59. 3.1.5

Membarth R, Hannig F, Teich J, Korner M & Eckert W (2011). Frameworks for gpu accelerators: A comprehensive evaluation using 2d/3d image registration In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pp. 78–81. IEEE. 4

Miller AK, Marsh J, Reeve A, Garny A, Britten R, Halstead M, Cooper J, Nickerson DP & Nielsen PF (2010). An overview of the cellml api and its implementation. *BMC Bioinformatics* **11**, 178. 5.1

Murray CJ & Lopez AD (1997). Alternative projections of mortality and disability by cause 1990–2020: Global burden of disease study. *The Lancet* **349**, 1498–1504. 1

NumPy (2014). Numpy reference http://docs.scipy.org/doc/numpy-1.8.1/reference/ Accessed: 2014-04-24. 4.1.4

NVIDIA Corporation (2012). Nvidia's next generation cuda™ compute architecture: Kepler™ gk110 <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf> Accessed: 2014-04-05. 4.1

NVIDIA Corporation (2014a). Cuda toolkit documentation v6.0 <http://docs.nvidia.com/cuda/> Accessed: 2014-06-10. 4.1.1, 4.1, 4.1.1, 4.1.2, 9.2, 10.1, 14.2

NVIDIA Corporation (2014b). Nvidia cuda compiler driver nvcc <http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/> Accessed: 2014-07-18. 8.10

NVIDIA Corporation (n.d.). Nvidia® cuda® parallel programming and computing platform <http://www.nvidia.com/object/cuda_home_new.html> Accessed: 2014-04-09. 4, 4.1

Rush S & Larsen H (1978). A practical algorithm for solving dynamic membrane equations. *Biomedical Engineering, IEEE Transactions on* pp. 389–392. 3.1.5

Sundnes J, Artebrant R, Skavhaug O & Tveito A (2009). A second-order algorithm for solving dynamic cell membrane equations. *Biomedical Engineering, IEEE Transactions on* **56**, 2546–2548. 3.1.5

Sundnes J, Lines GT, Cai X, Nielsen BF, Mardal KA & Tveito A (2006a). *Computing the Electrical Activity in the Heart* Springer. 2.2, 2.2

Sundnes J, Nielsen BF, Mardal KA, Cai X, Lines GT & Tveito A (2006b). On the computational complexity of the bidomain and the monodomain models of electrophysiology. *Annals of biomedical engineering* **34**, 1088–1097. 2.2

SymPy (2014). Sympy documentation <http://docs.sympy.org/latest/index.html> Accessed: 2014-06-22. 5.2

Ten Tusscher K & Panfilov A (2006). Alternans and spiral breakup in a human ventricular. *Am J Physiol Heart Circ Physiol* **291**, H1088–H1100. 2.1, 2.2, 2.3, 9.1, 14.1, 14.2

Torelli L (1989). Stability of numerical methods for delay differential equations. *Journal of Computational and Applied Mathematics* **25**, 15–26. 3.1.2

Valdmanis FH (2012). Gpu accelerating the fenics project Master's thesis, University of Oslo. 4.1.1, 4.1.3

Vigueras G, Roy I, Cookson A, Lee J, Smith N & Nordsletten D (2014). Toward gpgpu accelerated human electromechanical cardiac simulations. *International journal for numerical methods in biomedical engineering* **30**, 117–134. 6.2

Winslow RL, Rice J, Jafri S, Marban E & O'Rourke B (1999). Mechanisms of altered excitation-contraction coupling in canine tachycardia-induced heart failure, ii model studies. *Circulation Research* **84**, 571–586. 1