|  |
| --- |
|  |
| Thesis for Master of Engineering |
|  |
| Enhancing the efficiency of animal-alternative *in-silico* drug cardiotoxicity prediction through CUDA-based parallel processing |
| December 2024 |
|  |
| Graduate School  Kumoh National Institute of Technology |
|  |
| Department of IT Convergence Engineering |
|  |
| Iga Narendra Pramawijaya |
|  |

|  |
| --- |
| Enhancing the efficiency of animal-alternative *in-silico* drug cardiotoxicity prediction through CUDA-based parallel processing |
|  |
| Supervisor Ki Moo Lim |
|  |
| This Thesis Presented for the Master of Engineering |
| December 2024 |
|  |
| Graduate School  Kumoh National Institute of Technology |
|  |
| Department of IT Convergence Engineering |
|  |
| Iga Narendra Pramawijaya |

|  |
| --- |
|  |
| Enhancing the efficiency of animal-alternative *in-silico* drug cardiotoxicity prediction through CUDA-based parallel processing |
|  |
| Iga Narendra Pramawijaya |
|  |
| Department of IT Convergence Engineering,  Graduate School  Kumoh National Institute of Technology |
|  |
| Abstract |
|  |

Introduction: This research focuses on enhancing *in-silico* cardiotoxicity prediction by utilising GPU-based parallel computing. Traditional CPU-based simulations are computationally expensive, especially for large-scale studies. By leveraging CUDA programming, we aim to optimise simulation efficiency while maintaining the accuracy of cellular electrophysiological models.

Method: The study employed three well-established cardiac cell models: ORd 2011, ORd 2017, and ToR-ORd. Simulations were conducted using GPU-based implementations of ordinary differential equation (ODE) solvers, with the Rush-Larsen method applied for ORd 2011 and a Forward Euler approach for ORd 2017 and ToR-ORd. The simulations were validated against CPU-based OpenCOR results, with performance evaluated in both drug-free and drug-induced conditions.

Results: GPU simulations demonstrated equivalent accuracy to CPU-based results, effectively replicating action potential dynamics and key biomarkers across all cell models. However, the Forward Euler solver required more computation time compared to the Rush-Larsen method. Computational performance analysis revealed significant efficiency improvements in GPU-based simulations, particularly in handling large-scale datasets.

Conclusion: This research successfully validates GPU-based parallel computing as a reliable and efficient approach for *in-silico* cardiotoxicity prediction. The findings support its potential for accelerating drug discovery processes while reducing reliance on animal testing. Future work will focus on expanding model complexity and variabilities to further enhance the system’s applicability

|  |
| --- |
|  |

|  |
| --- |
| CUDA기반 병렬처리를 통한 동물대체 인실리코 약물 심독성 예측 효율성 증대 |
|  |
| Iga Narendra Pramawijaya |
|  |
| 금오공과대학교 대학원 IT융복합공학과 |
|  |
| 요 약 |
|  |

소개 : 본 연구는 GPU 기반 병렬 컴퓨팅을 활용하여 *in-silico* 심장 독성 예측을 향상시키는 데 초점을 맞추고 있습니다. 기존의 CPU 기반 시뮬레이션은 대규모 연구에서 계산 비용이 높아 비효율적입니다. CUDA 프로그래밍을 활용하여 세포 전기생리학 모델의 정확성을 유지하면서 시뮬레이션 효율성을 최적화하는 것을 목표로 합니다.

방법 : 본 연구에서는 ORd 2011, ORd 2017, ToR-ORd 모델이라는 세 가지 잘 확립된 심장 세포 모델을 사용했습니다. ODE(상미분방정식) 해석기를 GPU 기반으로 구현하였으며, ORd 2011에는 Rush-Larsen 방법을, ORd 2017 및 ToR-ORd에는 Forward Euler 방법을 적용했습니다. 시뮬레이션 결과는 CPU 기반 OpenCOR 결과와 비교하여 검증하였으며, 약물 없는 상태와 약물 유도 상태 모두에서 성능을 평가했습니다.

결과 : GPU 시뮬레이션은 모든 세포 모델에서 CPU 기반 결과와 동일한 정확성을 보였으며, 활동 전위 역학 및 주요 바이오마커를 효과적으로 재현했습니다. Forward Euler 방법은 Rush-Larsen 방법에 비해 계산 시간이 더 오래 걸렸습니다. 계산 성능 분석에서는 특히 대규모 데이터셋 처리에서 GPU 기반 시뮬레이션이 상당한 효율성 개선을 보여주었습니다.

결론 : 이 연구는 GPU 기반 병렬 컴퓨팅이 신뢰할 수 있고 효율적인 인실리코 심장독성 예측 방법임을 성공적으로 입증했습니다. 연구 결과는 약물 개발 과정을 가속화하고 동물 실험 의존도를 줄이는 데 기여할 가능성을 뒷받침합니다. 향후 연구에서는 모델의 복잡성과 변동성을 확장하여 시스템의 적용 가능성을 더욱 향상시키는 데 중점을 둘 것입니다.

**Contents**

[[List of Figures] iii](#_Toc183513890)

[[Glossary] iv](#_Toc183513891)

[[Acknowledgement] v](#_Toc183513892)

[Chapter 1. Introduction 1](#_Toc183513893)

[1.1 *In-Silico* Electrophysiology Simulation 1](#_Toc183513894)

[1.2 Parallel Computing 2](#_Toc183513895)

[1.2.1 Central Processing Unit (CPU) for Parallel Computing 3](#_Toc183513896)

[1.2.2 Graphics Processing Unit (GPU) for Parallel Computing 4](#_Toc183513897)

[1.2.3 CUDA 4](#_Toc183513898)

[1.2.4 CellML 5](#_Toc183513899)

[1.3 Previous Study 6](#_Toc183513900)

[1.4 Objectives 7](#_Toc183513901)

[Chapter 2. Methodologies 8](#_Toc183513902)

[2.1 Generating C Code from CellML 8](#_Toc183513903)

[2.2 GPU Memory Adjustment and Offseting 9](#_Toc183513904)

[2.3 Solving Ordinary Differential Equations 12](#_Toc183513905)

[2.4 Simulation Protocol and Code Organisation 14](#_Toc183513906)

[2.5 Output Format 16](#_Toc183513907)

[2.6 Compilation, Input Files, and Testing 17](#_Toc183513908)

[2.6.1 Compiling with makefile 17](#_Toc183513909)

[2.6.2 Required Input Files 18](#_Toc183513910)

[2.6.3 Testing and Result Validation Method 20](#_Toc183513911)

[Chapter 3. Results and Discussion 21](#_Toc183513912)

[3.1 GPU Simulation Result Using ORd 2011 Cell Model 21](#_Toc183513913)

[3.1.1 Result Validation 22](#_Toc183513914)

[3.1.2 Result Validation Under Drug 23](#_Toc183513915)

[3.1.3 Computational Time and Efficiency Analysis 24](#_Toc183513916)

[3.2 GPU Simulation Result Using ORd 2017 Cell Model 25](#_Toc183513917)

[3.2.1 Result Validation 25](#_Toc183513918)

[3.2.2 Result Validation Under Drug 26](#_Toc183513919)

[3.2.3 Computational Time and Efficiency Analysis 27](#_Toc183513920)

[3.3 GPU Simulation Result Using ToR-ORd Cell Model 28](#_Toc183513921)

[3.3.1 Result Validation 29](#_Toc183513922)

[3.3.2 Result Validation Under Drug 29](#_Toc183513923)

[3.3.3 Computational Time and Efficiency Analysis 30](#_Toc183513924)

[Chapter 4. Conclusion and Limitation 32](#_Toc183513925)

[4.1 Conclusion 32](#_Toc183513926)

[4.2 Suggestions 32](#_Toc183513927)

[[References] 33](#_Toc183513928)

[Appendix 35](#_Toc183513929)

[A. Project Structure 35](#_Toc183513930)

[B. Root folder 36](#_Toc183513931)

[a. Makefile 37](#_Toc183513932)

[b. .gitignore 39](#_Toc183513933)

[c. main.cu 40](#_Toc183513934)

[d. test\_compile.bat 64](#_Toc183513935)

[C. ‘bin’ Folder 64](#_Toc183513936)

[a. CVAR 64](#_Toc183513937)

[b. Control 64](#_Toc183513938)

[c. drug 65](#_Toc183513939)

[d. result 65](#_Toc183513940)

[D. ‘cellmodels’ Folder 65](#_Toc183513941)

[a. Ohara\_Rudy\_2011.hpp 66](#_Toc183513942)

[b. Ohara\_Rudy\_2011.cpp 67](#_Toc183513943)

[c. Cellmodel.hpp 88](#_Toc183513944)

[d. enums/enum\_Ohara\_rudy\_2011.hpp 89](#_Toc183513945)

[E. ‘modules’ Folder 95](#_Toc183513946)

[a. cipa\_t.cu and cipa\_t.cuh 95](#_Toc183513947)

[b. drug\_conc.cpp and drug\_conc.hpp 97](#_Toc183513948)

[c. glob\_funct.cpp and glob\_funct.hpp 98](#_Toc183513949)

[d. glob\_type.cpp and glob\_type.hpp 101](#_Toc183513950)

[e. gpu.cu and gpu.cuh 101](#_Toc183513951)

[f. param.cpp and param.hpp 109](#_Toc183513952)

[F. Critical Scripts 110](#_Toc183513953)

[a. Main script 110](#_Toc183513954)

[b. GPU control script 111](#_Toc183513955)

[c. Parameters 112](#_Toc183513956)

[G. Commands and Flag Usages 112](#_Toc183513957)

[H. Troubleshooting 114](#_Toc183513958)

# [List of Figures]

[[Figure 2. 1] OpenCOR interface when selecting ToR-ORd model and converting it to C codes on MacOS. 9](#_Toc183513874)

[[Figure 2. 2] Main difference in values storing paradigm after CUDA-parallelisation, assuming column size is 13. 11](#_Toc183513875)

[[Figure 3. 1] Action Potential (mV) Shape of both CPU (blue) and GPU (orange) Result Using ORd 2011 22](#_Toc183513876)

[[Figure 3. 2] Action Potential Shape (mV) of both CPU (dashed) and GPU under drug effect Using ORd 2011 23](#_Toc183513877)

[[Figure 3. 3] Simulation time comparison between GPU and CPU in ORd 2011 24](#_Toc183513878)

[[Figure 3. 4] Action Potential (mV) Shape of both CPU (blue) and GPU (orange) Result Using ORd 2017 25](#_Toc183513879)

[[Figure 3. 5] Action Potential Shape (mV) of both CPU (dashed) and GPU under drug effect Using ORd 2017 26](#_Toc183513880)

[[Figure 3. 6] Simulation time comparison between GPU and CPU in ORd 2017 28](#_Toc183513881)

[[Figure 3. 7] Action Potential (mV) Shape of both CPU (dashed blue) and GPU (orange) Result Using ToR-ORd 29](#_Toc183513882)

[[Figure 3. 8] Action Potential (mV) Shape of both CPU (dashed) and GPU under drug effect Using ToR-ORd cell model 30](#_Toc183513883)

[[Figure 3. 9] Simulation time comparison between GPU and CPU in ToR-ORd cell model 31](#_Toc183513884)

# [Glossary]

|  |  |
| --- | --- |
| CUDA | Compute Unified Device Architecture |
| CiPA | Comprehensive *in vitro* Proarrhythmia Assay |
| CPU | Central Processing Unit |
| GPU | Graphics Processing Unit |
| MPI | Message Parsing Interface |
| ODE | Ordinary Differential Equations |
| GP-GPU | General Purpose computing on Graphics Processing Unit |
| CaI | Intracellular calcium concentration. |
| IKr | Rapid delayed rectifier potassium current. |
| hERG | Human Ether-à-go-go-Related Gene |
| ICaL | L-type calcium current |
| IK1 | Inward rectifier potassium current |
| IKs | Slow delayed rectifier potassium current |
| Ito | Transient outward current |
| INaL | Late sodium current |
| INa | Sodium current |
|  |  |
|  |  |
|  |  |
|  |  |

# Introduction

Cardiovascular diseases are the leading global causes of death, which emphasizes the importance of effective methods for drug discovery. Traditionally, drug cardiotoxicity prediction is achieved using animal testing, which is controversial and has several drawbacks. Modern *in-silico* or computer-based methods for drug cardiotoxicity prediction show promising results as an animal-alternative alternative. Nevertheless, some of them are computationally inefficient due to large amount of sample it needs to compute, to mimic natural variations. As the sample size increases, the complexity of the calculations grows, resulting in longer processing times and reduced efficiency. This limitation makes it difficult for traditional computational approaches to handle large-scale simulation (such that uses multi-sample scenario or inter-individual variations) within a reasonable timeframe. This research introduces an updated solution to address the computational inefficiencies of current *in-silico* drug cardiotoxicity simulations. By implementing Nvidia’s CUDA (Compute Unified Device Architecture)-based parallel programming on Graphics Processing Units (GPU) [1], our method significantly accelerates overall computational process, enabling faster handling of large-scale simulations. By leveraging the power of parallel processing, this approach not only enhances the *in-silico* simulation but also ensures that drug toxicity evaluations are both more practical and accurate, paving the way for broader and more ethical applications in real-world drug testing.

## *In-Silico* Electrophysiology Simulation

Biological *In-Silico* simulation is a field in computational biology in means of using the aid of computing devices to conduct mathematical calculations that accurately simulates cardiac responses within different conditions. Electrophysiology is a study of electrical activity in the heart, and it can be explained using mathematical model in form of ordinary differential equations (ODE). By studying how cardiac cell responses through its electrical activity, we can explain various phenomenon in detail such as cardiovasicular disesases, how effective a drug is, and how toxic a drug that is not meant for the heart cell is, to the heart cell. Furthermore the toxicity of a drug to the heart cell will be mentioned as cardiotoxicity. Simpy, *in-silico* electrophysiology simulation is a powerful computational tool used to model and study the electrical activity of the heart. These simulations provide valuable insights into cardiac function, disease mechanisms, potential treatments and what might harmful to cardiac function with a minimum invasive approach to collect data.

## Parallel Computing

Over recent decades, parallel computation has promised to accelerate overall computation speed. Parallel computing, from a technical standpoint, means performing many calculations simultaneously, based on the principle that large problems can often be divided into smaller tasks that can be processed at the same time. For programmers, the main challenge is how to allocate these concurrent tasks across multiple computing resources such as cores or even computers.

Parallel computing has both hardware and software requirements that are deeply interlinked. Computer hardware architechture must ensure it has more than one computing core, while parallel programming designs code utilizing more than one computing core. The hardware aspect of computer architecture supports parallelism by providing an infrastructure that can handle multiple, simultaneous processes or threads. Meanwhile, parallel programming focuses on efficiently using this hardware to perform tasks concurrently. This programming paradigm involves mapping tasks to these available cores to achieve simultaneous execution, ensure every core runs in harmony, and arranging output from each cores. When writing non-parallel programs, understanding the computer architecture is less crucial. However, in multi-core programming, a solid understanding of multicore architectures becomes essential for developing efficient and correct parallel programs.

The effectiveness of parallel computing depends on overcoming challenges like coordinating tasks, managing data dependencies, and distributing work evenly among computing units. Tasks that rely on each other must be executed in the right order to minimise idle time. Data dependencies, where one task requires the output of another, necessitate careful coordination to avoid delays or errors. Ensuring that all computing units are working efficiently is crucial to prevent bottlenecks. Addressing these issues requires both sophisticated programming techniques and well-designed systems.

### Central Processing Unit (CPU) for Parallel Computing

For decades, one key method of improving consumer computing performance was to increase the CPU’s clock speed. However, due to power, heat, and physical limitations in transistor miniaturization, this approach has reached its limits. As a result, manufacturers have shifted their focus from boosting clock speed to increasing the number of cores per CPU, inspired by supercomputers that achieve high performance by using large numbers of processors. Rather than relying solely on single-core performance, adding multiple cores also allows personal computers to improve processing power without clock speed increases. Widely-known standards to do parallel processing with a GPU is to use Open Multi-Processing (OpenMP) or the Message Passing Interface (MPI).

OpenMP (Open Multi-Processing) is a parallel programming model tailored for shared-memory architectures, commonly used to leverage the capabilities of multicore CPUs. By integrating simple compiler directives (pragmas) into C, C++, or Fortran code, OpenMP enables developers to parallelize loops and code sections with minimal changes [2]. Its approach involves dividing tasks among multiple threads that access a shared memory space, simplifying data management and parallel execution. Due to its ease of implementation, OpenMP is an optimal choice for high-performance applications running on single multicore processors, where shared memory facilitates efficient data handling.

MPI (Message Passing Interface), on the other hand is a standard designed for parallel computing in distributed-memory environments, such as clusters or supercomputers. MPI enables communication between processes operating in separate memory spaces by exchanging messages, making it suitable for applications requiring parallel execution across multiple networked nodes [3]. Although MPI introduces complexity due to explicit data management requirements, it provides exceptional scalability and flexibility. This makes it the preferred solution for high-performance computing tasks that demand extensive data communication between distributed systems.

OpenMP and MPI are two frameworks that serve distinct yet complementary roles in parallel computing. OpenMP is well-suited for shared-memory systems with straightforward parallelism, while MPI offers the scalability required for distributed systems that span multiple nodes. In this research, CPU parallelisation will use MPI from OpenMPI due to its multiple nodes capability [3]. CPU parallelisation will be compared with GPU parallelisation in terms of computational time only.

### Graphics Processing Unit (GPU) for Parallel Computing

Graphics processing unit initially designed to compute graphical calcucations that is repetitive, relatively more simple compared to what CPU calculates, but quantitatively much more calculations compared to CPU. Unlike CPUs, which are optimized for a wide variety of tasks and tend to have a smaller number of powerful cores, GPUs have thousands of smaller cores designed for high-throughput parallelism. This architecture allows GPUs to perform many calculations simultaneously, making them highly efficient for tasks that can be broken down into smaller, identical operations.

Beside of graphics computing, GPUs are also able to accelerate other computing purposes such as scientific simulation and machine learning. This demand creates new sub-field in computer programming, called GP-GPU programming, that stands for general-purposed graphics processing unit programming. Its high-throughput parallelism makes GPU suitable for scientific calculation that has massive datasets or extensive matrix calculation. With frameworks like CUDA and OpenCL, programmers can leverage GPU architectures to perform GP-GPU.

### CUDA

CUDA, short for Compute Unified Device Architecture, is a parallel computing platform and programming model developed by NVIDIA. It enables developers to harness the immense computational power of NVIDIA GPUs for general-purpose processing tasks [1]. CUDA is built on the foundation of extending standard C/C++ programming with GPU-specific features, making it accessible for developers already familiar with these languages. It provides tools that allow fine-grained control over GPU resources, enabling efficient parallel execution of computationally intensive tasks across thousands of GPU cores.

The computational model of CUDA is structured around a hierarchical organization of cores, blocks, and threads. At the highest level, the GPU consists of multiple streaming multiprocessors (SMs), each containing numerous CUDA cores. These cores execute the smallest unit of work in CUDA, referred to as a thread. Threads are grouped into blocks, which can contain hundreds or thousands of threads, depending on the GPU's architecture. Blocks are further organized into a grid, creating a hierarchy that allows the distribution of computational tasks across the entire GPU [1].

This hierarchy is essential for mapping complex problems onto the GPU efficiently. Developers can define the number of threads per block and the number of blocks per grid based on the problem's computational requirements and the GPU's hardware limitations. Each thread has a unique thread ID within its block, and each block has a unique block ID within the grid. Using these IDs, CUDA programs can assign specific tasks or data portions to individual threads, ensuring even workload distribution.

The flexibility of CUDA's core-block-thread hierarchy enables scalability and efficiency. By tuning the grid and block dimensions, developers can optimize memory usage and parallelism for diverse applications, from image processing to numerical simulations. This design, combined with CUDA's extensive library ecosystem and advanced debugging tools, makes it a powerful platform for unlocking the full potential of GPU-based computing.

### CellML

CellML is an XML-based language created to represent mathematical models in a platform-independent format, facilitating model sharing between researchers and secure archival in repositories [4]. This standardization in a machine-readable form is essential in bioinformatics, as it enhances scientific accuracy, accelerates model development, and enables the integration of multiple models into complex, combined systems [5]. CellML supports collaboration by allowing models to be easily exchanged and archived. Several public databases host extensive collections of CellML models, with the CellML Model Repository being one of the most prominent. Additionally, the BioModels database includes models converted from the Systems Biology Markup Language (SBML) into CellML, broadening accessibility and compatibility for researchers using these bioinformatics resources. [6][7].

## Previous Study

Parallelisation in computational biology is not an entirely new concept. The Cells in Silico (CiS) framework presented by Berghoff et al. [8] offers a tool for simulating the growth and development of biological tissues. The modular and parallel design of CiS allows for flexible configuration of different model assumptions, making it applicable to a wide range of research questions. As demonstrated by the example of a 10003 voxel-sized cancerous tissue simulation at sub-cellular resolution, CiS can be used to explore complex biological processes at a high level of detail.

Utilisation of GPU in biological cell computing has been explored in previous researches. One of them is from Martinez, et al [9] in 2020. Miguel, et al. explored an adaptive parallel simulator to solve performance loss in massive parallel membrane computing devices known as membrane systems or P systems. The paper demonstrates the effectiveness of this approach by extending an existing simulator for Population Dynamics P systems. Experimental results show that this adaptive simulation can significantly improve performance, up to 2.5x on both GPUs and multicore processors.

Related to drug toxicity and discovery, other researchers tried to approach and optimise drug development process using parallel computing approach as well. Previously, McIntosh-Smith et, al. developed *in-silico* drug screening method on multiple core processors. McIntosh-Smith et, al. developed BUDE (Bristol University Docking Engine), a drug discovery tool, simulating molecular docking. To speed up calculations on powerful processors with multiple cores, BUDE has been adapted to work with OpenCL, a common language for parallel programming [10]. As a result, McIntosh-Smith et, al. achieved of 46% at peak, or 1.43 TFLOP/s on a single Nvidia GTX 680.

Amar et, al. developed a parallelisation on biochemical simulation of metabolic pathways in their high level computational simulation. This method allows Barth et, al. to run simulations with more complex models, featuring a greater number of chemicals and reactions. Hence, Barth et, al. can achieve more realistic, lifelike outcomes while using less computing time [11].

Furthermore, advancements in computer hardware and parallel processing techniques have significantly improved the speed and efficiency of these heart simulations. This technological progress allows researchers to analyze larger datasets and more complex models at an unprecedented pace, making computer simulations an essential tool in contemporary heart research. As computer technology continues to advance, computer-based heart simulations are poised to play an even more crucial role in uncovering new insights into heart function and developing targeted treatments for heart diseases.

Recent studies highlight the growing use of *in-silico* approaches to investigate specific cardiac conditions and evaluate pharmacotherapies. For example, Whittaker et, al. used computational modeling to assess the effects of mutations associated with Short QT Syndrome and their impact on atrial arrhythmias. Their findings illustrate how *in-silico* simulations can explore drug responses and guide pharmacological strategies in addressing genetic cardiac disorders​ [12].

## Objectives

The objective of this study is to enhance the efficiency and scalability of *in-silico* simulations for predicting drug cardiotoxicity by leveraging CUDA-based parallel processing. The research aims to:

1. Address computational bottlenecks caused by increasing sample sizes and complex calculations in traditional methods.
2. Optimize GPU resources for faster, large-scale simulations without compromising accuracy.
3. Validate the accuracy and reliability of GPU-based simulations compared to CPU-based methods.
4. Develop a practical and cost-effective approach suitable for real-world drug discovery applications, reducing reliance on animal testing.

# Methodologies

Throughout this chapter, the author explains how to create sample-based parallelisation cardiac cell simulation platform on GPU, and parallelise CellML based *in-silico­* cell model. Each subchapter will explain in detail development and code conversion process in this research. At the end, a software engineer or other researcher can replicate the parallelisation concept and utilise this base to more multi-sample *in-silico* simulation.

## Generating C Code from CellML

In order to generate codes in various programming language based on CellML’s XML, we can use various third party libraries or application. One of the most popular application is OpenCOR. OpenCOR is a versatile software tool designed for modeling and simulation of biological processes, including those described in CellML. It provides functionality to parse CellML models and convert them into executable code in various programming languages, including C. In addition to generating C code, OpenCOR supports model editing, running simulation, and analysis, making it a comprehensive platform for working with CellML models. Its extensibility and integration with other tools, such as Python scripting, further enhance its utility for researchers. By leveraging OpenCOR, users can streamline the process of implementing CellML models into broader computational workflows, such as those involving high-performance computing or *in-silico* drug testing. Results from this GPU modification will also validated using OpenCOR’s result.

This research involves three different cell models. O’Hara-Rudy 2011 [13], 2017 [14] and ToR-ORd [15]. After installing OpenCOR, open the application and search for these three cell models in the search bar on top left corner (PMR). Select file with .cellml extension. Select tools, and export the CellML file to C code. Figure 2.1 shows the OpenCOR GUI on MacOS.



[Figure 2. 1] OpenCOR interface when selecting ToR-ORd model and converting it to C codes on MacOS.

Having the C code of the cell model is important because CUDA programming uses .cu format, that is similar to C. CUDA is built upon C/C++, extending it for GPU programming. Both use C syntax as their base language, many fundamental syntax elements like loops, conditionals, functions, and function definitions are similar in both CUDA and C. CUDA programs include host code that runs on the CPU, which is essentially standard C/C++ code. For programmers familiar with C/C++, CUDA maintains a relatively low learning curve by building upon familiar concepts, only adding memory allocation and parallel processing paradigms.

## GPU Memory Adjustment and Offseting

Efficient memory management is critical in GPU programming, as the performance of a CUDA-based application heavily depends on how data is transferred between the host (CPU) and device (GPU), as well as how it is organized within the GPU's memory. The GPU has several memory types, including global, shared, constant, and local memory, each with unique characteristics and access speeds. Proper adjustment and offsetting of these memory types can significantly enhance computational efficiency. Due to its unique characteristics, code development in this research requires a dedicated GPU unit. We develop the code using a commercially available gaming-grade specification personal computer with GPU specifically from NVIDIA only. The code will not be compatible with GPU other than NVIDIA. Details of the GPU specification will be mentioned in chapter 3 and appendix.

This research require us to choose the balance core (thread) per block. When choosing the number of cores per block in CUDA programming, several factors need to be considered to optimize performance and efficiency. CUDA executes threads in groups called warps, which consist of 32 threads [16]. Using a block size that is a multiple of 32 ensures that all warps are fully utilized, minimising idle threads and maximizing efficiency. We found that this configuration is not transferable across different GPUs. As the time writing this, NVIDIA 40xx series GPU supports 32 core per block, while older series like the 30xx only support 16 core per block. This generation also uses wraps [16], but it has less computing core, so we follow by using 16 as 16 times 2 is 32. Core per block adjustment also influenced by how many samples in one simulation. Each simulation sample count may vary, usually comes in the multiplication of 2000 (2000, 4000, etc.). Using the information of wraps usage, and have to choose an optimal number of core per block that can divide 2000 without any remainder, we choose 20 core per block as the most optimum after trial and error. This adjustment is applied to ensure that every sample has its own computing core, maximising sample based parallelisation. By default, our code will select 32 as its core per block number. Error may raised in that configuration due to hardware limitation, hence we can switch to 20 core per block to ensure the parallelisation process. Since 32 is not fully divisible by 2000, our code ensure there will always be more available cores (at least one core) for each sample by rounding up the number of block used. Details on this configuration will be attached in the Appendix.

Offsetting refers to managing data indexing to optimize memory access patterns. For example, ensuring that thread indices correspond to contiguous memory addresses can reduce memory bank conflicts and improve overall performance. Proper offset calculation is also crucial when dividing large datasets across multiple threads and blocks, ensuring each thread processes its designated segment efficiently and correctly. In this research, offseting implemented to simplify any multi-dimentional input. In order to enhance the efficiency of drug cardiotoxicity prediction, utilising multi-sample simulation is crucial. Cell model code from OpenCOR is only designed for single sample simulation, then conversion should be applied. On the other side, simplicity from the OpenCOR should be maintained. Also in the previous iteration of drug toxicity *in-silico* simulation based on CPU, it uses struct to temporarily store simulation results. In CUDA programming, there is no native multidimensional vector type like in higher-level programming languages. This research simplified all multi dimentional vector used in the previous iteration into 1 dimentional (1D) array. Offsetting is mainly used for pointing the correct data in a flattened 1D array.

Since CUDA natively operates on linear memory, storing data in a flattened 1D format aligns well with the GPU's memory architecture, allowing for optimal performance while maintaining simplicity. In this method, a multidimensional array is represented as a single contiguous block of memory, and elements are accessed using calculated indices. For example, a 2D array with dimensions (rows, columns) can be indexed as index = (current row \* row size) + columns. This linearization simplifies memory allocation and transfer between the host and device, ensuring compatibility with CUDA's memory management functions. Figure 2.2 explains graphically the flattening process and how multi-dimentional vector data differentiate samples (called as sample id) in the previous iteration of drug toxicity *in-silico* simulation.



[Figure 2. 2] Main difference in values storing paradigm after CUDA-parallelisation, assuming column size is 13.

The previous iteration of the code uses row to indicates different samples. Each sample will have their own identification number called the sample\_id, so each row correlates to one sample\_id. In the GPU version, instead of using rows to differentiate samples, it will utilise fact that each arrays has same number of columns. For example, array STATES has 43 columns, then STATES[0] up to STATES[42] is reserved for sample\_id = 0, STATES[43] up to STATES[84] is reserved for sample\_id = 1, and so on. Adapting with this approach, the current index can be determined by knowing row dimention, sample\_id and the number of specific column we want to select.

In the code from OpenCOR, most of the constants declarations, states calculations, rates calculations and algebraic formulations will be delivered in this form:

CONSTANTS[5] = 8314;

Or, for algebraic formulation

ALGEBRAIC[3] = 1.00000/(1.00000+exp((STATES[0]+87.6100)/7.48800));

At the top of the code from OpenCOR, we can see how many CONSTANTS, STATES (similar to number of RATES), and ALGEBRAIC are there in the code. It looks like:

There are a total of 223 entries in the algebraic variable array.

There are a total of 43 entries in each of the rate and state variable arrays.

There are a total of 163 entries in the constant variable array.

These are going to be the row size, while number of sample will be the current row. Number of sample will be declared later in this chapter, we will name it ‘sample\_id’.

Apply offsetting in every CONSTANTS, STATES (similar to RATES), and ALGEBRAIC array occur in the code by applying this to every array index:

new\_index = (sample\_id \* row size) + columns.

Hence, all of the declarations and calculations in the code should look like:

CONSTANTS[(sample\_id \* 163) + 5] = 8314;

Or, for algebraic formulation

ALGEBRAIC[(sample\_id \* 223) + 3] = 1.00000/(1.00000+exp((STATES[(sample\_id \* 43) + 0]+87.6100)/7.48800));

Notice that the calculation for new index of each arrays are calculated within the declaration of each elements, and all arrays are treated the similar way.

## Solving Ordinary Differential Equations

The model relies on algebraic calculations and dynamic functions expressed in the form of ordinary differential equations (ODEs), which are essential for simulating the complex behaviors of biological systems. To efficiently solve these ODEs within a CUDA-based parallel processing framework, two distinct numerical methods were employed depending on the specific cell model: the Rush-Larsen method and a custom implementation of the forward Euler method. These methods were chosen to balance computational efficiency, numerical stability, and compatibility with the CUDA architecture.

For the ORd 2011 model, the Rush-Larsen method was utilized due to its computational efficiency and computational stability in this context. This method effectively integrates stiff components of the equations, making it well-suited for the dynamic features of the ORd 2011 model. This method was implemented with a dynamic time-stepping mechanism by adjusting the time step during each iteration, while balancing acceptable numerical errors [17]. Initially, we aimed to generalize the use of the Rush-Larsen method across all cell models. However, when applied to ORd 2017 and ToR-ORd models, this approach exhibited instability, failing to provide reliable results. To address this limitation, a simple forward Euler method was implemented for these models. While the Euler method produced accurate and stable results, particularly for the ORd 2017 and ToR-ORd models, it proved to be computationally intensive, significantly increasing the runtime.

To optimize the parallelisation process, we focused on simplifying the algorithm by enabling parallel threads to process multiple samples rather than multiple equations simultaneously. The Forward Euler solver also can be upgraded with a dynamic time-stepping mechanism, by adjusting the time step during each iteration. Despite the trade-off between computational speed and stability, this combination of methods ensures that the CUDA-based framework effectively supports the diverse requirements of different cell models while maintaining accuracy and to reduce numerical errors. This research implements the ODE solver inside the cellmodel code as a function. Forward Euler is a simple method to solve ODE in particular time. The forward Euler method calculates the next value of a variable by taking its current value (STATES array) and adding the product of the rate (RATES array) of change and the time step (*dt*). This straightforward approach makes the method computationally simple and easy to implement. Mathematically, it is expressed as:

*xn+1=xn+rate(xn)⋅Δt*

where *xn+1*​ is the current value, *rate(xn)* represents the rate of change at *xn*, and *Δt* is the time step. In the converted code we are working now, add a function to implement this calculation. This can be achieved by implementing for loop such as:

void solveEuler( double \*STATES, double \*RATES, double dt, int sample\_id)

{

for(int i=0;i<43;i++){

STATES[(43 \* sample\_id) + i] = STATES[(43 \* sample\_id) + i] + RATES[(43 \* sample\_id) + i] \* dt;

}

}

The function uses a for loop to iterate over the 43 state variables associated with a single sample in ToR-ORd cell model. For each state variable, it calculates the new value using the forward Euler method formula. The formula is applied to the corresponding state variable and rate of the selected sample, determined by the sample\_id. By multiplying sample\_id by 43 (the number of state variables per sample in ToR-ORd cell model), the function accesses the correct block of memory in the flattened 1D array for both STATES and RATES. This ensures that updates are sample-specific and do not interfere with other samples.

## Simulation Protocol and Code Organisation

The whole GPU simulation code repository consist of three main folders, bin, cellmodels, and modules. Bin folder stands for ‘binary’, means we are going to compile our code and put the executeable in this folder. This folder also has a folder for storing input data (named ‘drugs’ by default) and a folder named ‘result’ to collect all output from the simulation. This folder also contains a text file known as the input deck. Input deck holds simulation parameters we can change, without re-compiling the code. As some part of the output are hard-coded, we cannot rename or delete the ‘result’ folder, as it will result a segmentation fault or a crash at the end of the simulation.

The second folder is the cellmodels folder. This folder holds codes of cell models like ToR-ORd, ORd 2011, ORd 2017, or others. Each cell model also requires a header file, so it can be run from another code. This folder also contains cellmodel header (cellmodel.hpp) which contains common functions and variables used in each of the cell models. Some of commonalities in these cell models are: 1) they have arrays of different sizes that need to be accounted for the offsetting, 2) initialisation function (initConsts() function), 3) ODE solver, and 4) sometimes a dynamic time step function. As an addition to make identifying each gate, parameters, or variables in the cell model easier, we introduce enumeration as a header file. This header located inside the enums folder, in cellmodels folder. This header simply create enumerator for each index in each array. For example, in ToR-ORd cell model, CONSTANTS[1] is NaO (millimolar) in component extracellular. This header will enable CONSTANTS[1] aliased with CONSTANTS[nao] for easier tracing. Detail of all codes mentioned in this research will be attached in the Appendix.

Next, in the modules folder, there are a lot of utility codes stored in this folder. First, is the cipa\_t header, that define custom struct datatype that will store simulation biomarkers result. Then, there is global function script and header. Global function is used to read input deck, input flags, and some common variables. After that, there is global type script and header to declare custom data type to store drug data (IC50 and Hill fitting result). Next, we have script and header of parameters, named param.cpp and param.hpp. The parameters script acts as declarator to default simulation parameter, including the default input file directory. We provided a default input file to act as ‘failsafe’, to ensure simulation still runs for checking even if user did not provide any input. The last one but the most important, is the gpu.cu and gpu.cuh.

The file `gpu.cu` is central to all parallel processing in this research. It is specifically designed to handle computational tasks leveraging GPU acceleration, ensuring efficient parallel execution. Within `gpu.cu`, a key function named ‘kernel\_DrugSimulation’ manages the parallelisation process. This function is responsible for batching operations, determining the data to be processed, and managing memory sharing among threads. This primary function orchestrates the parallelisation by distributing tasks across thousands of GPU threads. Then, each thread runs the same function, the simulation function concurrently but processes a distinct portion of the data. The key function then calls the simulation function helper functions are called within this function, and these are effectively “multiplied” across the threads, each executing independently. This design ensures the workload is evenly distributed and processed simultaneously, fully utilising the computational power of the GPU. By encapsulating the core parallel processing logic within ‘gpu.cu’, the research achieves a streamlined and modular structure, making it easier to maintain, optimise the GPU-specific operations, and simplicity for future modifications.

The header for the key GPU code, ‘gpu.cuh’ help to declare three different functions, the key function, and two other simulation functions, named ‘kernel\_DoDrugSim’ and ‘kernel\_DoDrugSim\_single’. Similar to their names, ‘kernel\_DoDrugSim’ was designed to being runned multiple times, and ‘kernel\_DoDrugSim\_single’ was designed to run only for one or two times to collect necessary simulation result. The ‘kernel\_DoDrugSim’ will run for thousands of paces first, amplifying the drug effect in the simulation, then ‘kernel\_DoDrugSim\_single’ will run only once or twice (depending on the cell model) to capture and calculate important simulation results in the form of time series data and features called biomarkers. This research split the drug effect amplification and data capture process due to memory limitation. The method in ‘kernel\_DoDrugSim\_single’ will take up to 60% more memory to save all necessary information. Because ‘kernel\_DoDrugSim’ can save some memory usage, we can use the remain memory space to put more samples, trading off some feature calculation away. The output from ‘kernel\_DoDrugSim’ will be used as input for ‘kernel\_DoDrugSim\_single’, measuring the amplified drug effect after thousands of paces.

## Output Format

The simulation produces two distinct types of output files, a biomarker file and time-series data files, along with one intermediate cache file. The cache file is generated as the output of the `kernel\_DoDrugSim` function, which represents the initial phase of the simulation. During this phase, the function runs the simulation for thousands of cycles (referred to as paces) to amplify the drug effects within the model. After this initial phase, the `kernel\_DoDrugSim\_single` function is executed, which generates the biomarker file and the time-series data files. All output files are organised into a dedicated folder within the `result` directory for efficient storage and retrieval.

The biomarker file provides a summary of key features extracted from the simulation for each sample. It includes data such as sample numbers, qNet, qInward, and action potential shape analysis result. These biomarkers represent crucial physiological parameters simulated under drug influence, and they are instrumental for downstream analyses, such as machine learning-based predictions. The time-series file offer a detailed temporal view of each sample’s behaviour. Each sample has its own individual time-series file; thus, a simulation involving 2000 samples will result in 2000 time-series files. These files capture parameters such as time, action potential, voltage gradient over time, Cai, INa, INaL, ICaL, IKs, IKr, IK1, and Ito. Using this detailed data, it is possible to plot the drug-induced cellular responses over a single cycle, facilitating visualisation and deeper analysis of dynamic behaviours.

## Compilation, Input Files, and Testing

This subchapter will discuss about methods to compile the research’s code, required input files for the simulation and methods to ensure simulation result integrity by various testing. This research’s code repository consist of some C++ and CUDA codes needed to be compiled and linked before use. We utilised Makefile to compile all of the codes in order and link them together, as well with linking the main code of this research with CUDA’s system library. This subchapter will also discuss on required input files and how to declare them when running the simulation. At the end, we demonstrate how we test the code by comparing it with OpenCOR’s result and ensure compilation runs well.

### Compiling with makefile

This research’s code repository provided Makefile to outline a clear and structured approach to compiling the CUDA-based C++ scripts and headers. It incorporates key elements like specifying source files, dependencies, compilation flags, and cleaning commands. Key concepts, such as the use of := for immediate value assignment, are highlighted to ensure consistent behaviour during execution. Commentary further guides users on technical aspects. The Makefile for this research will be provided in the appendix.

The .PHONY directive specifies that all and clean are symbolic targets rather than actual files or directories. This ensures that Make does not mistake them for real entities during execution. The PROGNAME variable is used to define the final executable's name (drug\_sim), making it easy to change the output program's name if needed. These simple, high-level definitions contribute to a more modular and maintainable Makefile.

The Makefile specifies nvcc, NVIDIA’s CUDA compiler, as the compiler, ensuring compatibility with GPU-based computation. It also utilises nvlink for linking purposes. Key flags are defined for both compilation and linking. CPPFLAGS includes the path to CUDA header files, while CXXFLAGS incorporates options such as -Wall for warnings, -O2 for optimisation, and -fpermissive for handling relaxed syntax. The LDFLAGS variable defines linker flags, such as paths to CUDA libraries, GPU architecture specifications (-arch=sm\_86), and relocatable device code support (-rdc=true), ensuring optimal performance on the targeted GPU architecture.

To manage source and header files, the Makefile uses wildcard functions to automatically detect files with relevant extensions (e.g., .cpp, .cu, .c for source files and .hpp, .h, .cuh for headers). This approach ensures that the build process dynamically adapts to new files added during development, eliminating the need for manual updates to the Makefile. Object files are generated by substituting the .cpp extension with .o, ensuring a seamless mapping from source to object files.

The all target is set as the default goal, compiling the entire project by depending on the program name ($(PROGNAME)). The linking process combines all object files into the final executable using the specified compiler and linker flags. Explicit rules are defined for generating object files from source files, utilising CUDA-specific flags such as -x cu, -dc, and -arch=sm\_86 to ensure compatibility with GPU-based parallel processing.

A clean target is included to facilitate the removal of temporary files such as object files (\*.o) and the executable. This ensures a clean workspace for subsequent builds. The @ symbol suppresses the command echo, while the - symbol allows the process to continue even if errors occur, such as when files are missing. Additionally, the LIBS variable specifies external libraries like OpenBLAS and CUDA-specific libraries (-lopenblas, -lpthread, -lcudart, -lcublas) that are crucial for performing mathematical and parallel computations.

Overall, this Makefile is structured to handle complex dependencies, optimise for GPU-based execution, and maintain flexibility for evolving project requirements. Its modular approach, detailed flag definitions, and automated file management make it a robust tool for managing the compilation process in this drug simulation research project.

### Required Input Files

To execute the simulation, two essential input files are required: an input deck and a drug data file. These files serve as the foundation for configuring the simulation environment and defining the drug properties necessary for the analysis. Together, they ensure the simulator has the parameters and data needed to accurately model the effects of the drug on the cell samples.

The first file, known as the input deck, is a text file containing simulation parameters. This file specifies crucial settings, such as the number of simulation steps, time intervals (`dt`), cell model identifiers, and other key configurations that govern how the simulation is run. By modifying this file, researchers can adapt the simulation to different experimental conditions without altering the underlying code. The flexibility provided by the input deck allows for efficient experimentation and testing across a wide range of scenarios. By default, an input deck file contains:

* Basic\_Cycle\_Length (length of one cycle in millisecond) = 1000
* Number\_of\_Pacing (number of cycle) = 1000
* Simulation\_Mode = 0
* Celltype = 0 (type of cell we want to simulate) (0: endo, 1: epi, 2: M
* Is\_Dutta = 1 (means conductance scaling from Dutta et al. 2017)
  + Dutta’s conductance scaling may vary. ToR-ORd cell model do not require this.
* Is\_Post\_Processing = 0 (set 0 to use ‘kernel\_DoDrugSim’ or set 1 to use ‘kernel\_DoDrugSim\_single’, we run mode 0 first, then mode 1)
* Use\_Conductance\_Variability = 0 (1: read additional file which contain individual conductance variability)
* Pace\_Find\_Steepest = 250 (timing to start searching steepest dvdt repol. Means we start searching from last 250 cycles, or cycle number 750-1000) (minimum value: 2)
* Drug\_Name = quinidine
* Concentrations = 3237.0 (concentration of the drug in mMol)
* GPU\_Index = 0 (choose which GPU will run the simulation, a PC with 1 GPU should let it 0)

The second file is a CSV file containing drug-specific data, particularly the IC50 and Hill coefficient values. These pharmacological parameters are critical for modelling the drug's effect on ion channels and other cellular processes. The IC50 value represents the drug concentration at which 50% of its maximal inhibitory effect is observed, while the Hill coefficient describes the slope of the dose-response curve, indicating the cooperativity of drug binding. This file provides the simulator with the necessary data to simulate the drug’s interaction with the cells accurately.

These required input files declared by adding flags in the running parameter. After compilation, the compiled simulator will be available in the ‘bin’ folder. Then add two flags when running drug\_sim, -input\_deck declares the location of the input deck file, and -hill\_file declares the location of IC50 and hill file. By using these two files, the simulator integrates user-defined configurations with drug-specific properties, creating a dynamic and adaptable environment for simulating drug-induced cellular responses. The modularity of this input system ensures that new parameters or drugs can be tested efficiently, making the simulation framework highly versatile and scalable for diverse research needs.

### Testing and Result Validation Method

Testing and validating the results of the simulator is a critical step to ensure its accuracy and reliability. The process begins with compiling the code and resolving any errors that may arise during the compilation stage. This involves carefully examining the Makefile and codebase to ensure all dependencies are correctly linked and that no syntax or compatibility issues are present. Once the code is successfully compiled, it can then proceed to the testing phase.

The initial testing phase involves running the simulator for a limited number of iterations. This step aims to confirm that the program executes correctly without unexpected crashes or errors. During this phase, intermediate outputs are monitored to verify that the calculations align with expected values. Any discrepancies observed at this stage are investigated and resolved before moving forward.

Once the basic functionality is verified, the simulator undergoes a more rigorous testing phase by running a full GPU-based simulation of 1000 paces. This comprehensive test ensures that the GPU's parallel processing capabilities are functioning as intended and that the system can handle the computational load efficiently. The outputs of this simulation, including both biomarker files and time-series data, are then compared to the results obtained from running the same simulation in OpenCOR.

The comparison between the GPU-based simulator and OpenCOR serves as a key validation step. For the results to be considered accurate, the discrepancies in time-series data between the two methods should not be able to visually distinguished when stacked onto one plot. Any significant deviations are analysed to determine whether they result from numerical methods, precision differences, or potential errors in implementation. By following this systematic approach to testing and validation, the reliability of the simulator is established, providing confidence in its ability to produce accurate and meaningful results for drug-induced cellular response simulations.

# Results and Discussion

This chapter presents the findings of the GPU-based cellular simulation for three different cell models: ORd 2011, ORd 2017, and ToR-ORd. Each section provides a detailed analysis of the results obtained from the simulations, focusing on validating the outcomes, assessing the effects of drugs, and evaluating computational performance and efficiency.

The first section, outlines the results for the ORd 2011 cell model. The accuracy of the simulation is validated by comparing the GPU-generated outputs with benchmarks obtained from solvers in OpenCOR. The impact of drug-induced changes on cellular behaviour is then analysed in the next sub-section, followed by a discussion of the computational time and efficiency improvements achieved with GPU acceleration. Similarly, the second section, delves into the simulation results for the updated ORd 2017 cell model. The third section focuses on the results for the ToR-ORd cell model. As with the previous models, the accuracy of the simulation is validated, the drug-induced changes are analysed, and computational performance is reviewed. Together, these sections provide a comprehensive overview of the performance and reliability of GPU-accelerated cellular simulations across multiple models. This chapter not only demonstrates the feasibility of the approach but also underscores its potential for large-scale and rapid cardiotoxicity prediction in drug development. All control result obtained in no-drug situation, and the drug used when analysing drug-induced changes simulated under bepridil, with concentration of 33 mMol (cmax 1), 66 mMol (cmax 2), and 132 mMol (cmax 4). All result both with and without drug effect were run for 1000 pacing, and lasts for 1000 milliseconds.

## GPU Simulation Result Using ORd 2011 Cell Model

This section examines the results of GPU-based simulation using the ORd 2011 cell model. The ODE solver in ORd 2011 model was using the Rush-Larsen method which offers faster computational time by optimising the handling of gating variables in the equations. This approach not only accelerates simulations but also ensures sufficient numerical stability for this cell model.

### Result Validation

To validate the results of the GPU simulation, we compared the action potential outputs against reference solutions obtained from the OpenCOR. Visual comparisons of time-series plots for action potentials were performed to ensure qualitative agreement. Key electrophysiological biomarkers, such as action potential duration, calcium transient properties, and ionic currents, were also compared. These biomarkers were compared under a same, no drug conditions, ensuring that the GPU-based simulation accurately reproduces the physiological dynamics from the ORd 2011 model.

The findings revealed the GPU simulation result is almost exactly same with its CPU predecessor. Figure 3.1 shows visually action potential from both simulation platforms.

[Figure 3. 1] Action Potential (mV) Shape of both CPU (blue) and GPU (orange) Result Using ORd 2011

As shown, little to no difference from both of the result, indicating a valid result from the GPU-based simulation. Promising more efficient *in-silico* drug cardiotoxicity prediction.

### Result Validation Under Drug

This section evaluates the accuracy of the GPU-based simulation for the ORd 2011 cell model under drug conditions. Drug effects were modelled by adjusting ionic current parameters using IC50 and Hill coefficient values, applied consistently in both GPU and CPU (OpenCOR) simulations. Validation involved comparing action potential traces and key biomarkers, such as action potential duration (APD), calcium transients, and ionic currents.

[Figure 3. 2] Action Potential Shape (mV) of both CPU (dashed) and GPU under drug effect Using ORd 2011

The results from figure 3.2 showed that despite the added complexity of drug effects, the GPU simulation closely matched the outputs of the CPU-based OpenCOR simulation. This confirms the GPU implementation's reliability in replicating physiological and pharmacological responses under drug conditions.

### Computational Time and Efficiency Analysis

This analysis compares computational time between two hardwares. GPU based simulation executed using NVIDIA RTX 4090 with 24 GB of memory. CPU based simulation also uses parallel processing of 10 cores Intel Xeon (x86) Silver 4215 CPU @ 2.50 Ghz. The computational time compared for 8000 samples (each drug has 2000 samples, and 4 concentrations). In theory, GPU cores operate at lower clock speeds, making them inherently less powerful than CPU cores, which is why CPUs are typically preferred for single-sample simulations. The computation time for CPUs increases linearly with the sample size and pacing, meaning that as the number of samples grows, so does the time require for computation.

[Figure 3. 3] Simulation time comparison between GPU and CPU in ORd 2011

In contrast, GPU parallelisation eliminates this linear growth as seen in the figure 3.3. the time it takes to compute one sample is nearly the same regardless of how many samples are processed, due to its parallel computing architecture. GPU achieved a speedup of up to 40.91 times compared to a 10-core CPU. 10-core setups are more common in practice in CPU parallelisation, making GPU parallelisation significantly more time-efficient for most common and practical simulations.

## GPU Simulation Result Using ORd 2017 Cell Model

This section explores the outcomes of the GPU-based simulation using the ORd 2017 cell model. Unlike the ORd 2011 model, which leverages the computational efficiency of the Rush-Larsen method, the ORd 2017 model uses the forward Euler method for solving ordinary differential equations. While the forward Euler method is straightforward to implement and numerically stable for this model, it results in slower computational times compared to the Rush-Larsen method. This trade-off is necessary due to the instability observed when using the Rush-Larsen method with the ORd 2017 equations.

### Result Validation

Similar to previous, validation of GPU simulation results for the ORd 2017 cell model was conducted by comparing outputs with the reference solutions generated from OpenCOR. Visual comparisons of action potential time-series data confirmed a close alignment between the two simulation platforms.



[Figure 3. 4] Action Potential (mV) Shape of both CPU (blue) and GPU (orange) Result Using ORd 2017

Key biomarkers, such as action potential duration, were analysed under identical no-drug conditions to ensure the physiological fidelity of the GPU-based results. As with the ORd 2011 model, the findings demonstrated that the GPU simulation faithfully replicated the results from the CPU-based OpenCOR simulations. The numerical outputs showed no significant differences, confirming the reliability and accuracy of the GPU implementation for the ORd 2017 model. Figure 3.4 provides a visual comparison of the action potentials produced by the GPU and CPU simulations, illustrating their near-identical behaviour.

### Result Validation Under Drug

In this section, the accuracy of the GPU-based simulation for the ORd 2017 cell model was evaluated under drug conditions. The simulation incorporated drug-induced effects by modifying ionic current parameters based on IC50 and Hill coefficient values. These adjustments were applied uniformly across both the GPU and CPU (OpenCOR) simulations to ensure consistency in the drug response modelling. The validation process involved comparing action potential traces and key electrophysiological biomarkers, such as action potential duration (APD), calcium transient properties, and ionic current profiles, between the GPU and CPU simulations.

[Figure 3. 5] Action Potential Shape (mV) of both CPU (dashed) and GPU under drug effect Using ORd 2017

Despite the added complexity of drug effects, the GPU simulation produced outputs that were nearly identical to those of the CPU-based OpenCOR simulations, as shown in figure 3.5 . These findings confirm that the GPU implementation of the ORd 2017 model accurately captures the physiological and pharmacological responses of the cell model under drug conditions. This validation further establishes the robustness and reliability of the GPU-based simulation for scenarios involving drug effects.

### Computational Time and Efficiency Analysis

This section examines the computational performance of the GPU-based simulation for the ORd 2017 cell model, compared against a 10-core Intel Xeon (x86) Silver 4215 CPU @ 2.50 GHz. The GPU simulations were executed using an NVIDIA RTX 4090 with 24 GB of memory. Both hardware setups processed 8000 samples (2000 samples per drug at four different concentrations). The computation time on CPUs scales linearly with the number of samples due to sequential processing limitations, even when multiple cores are utilised. However, the GPU’s parallel processing architecture allows it to maintain consistent computational times regardless of sample size, effectively minimising linear growth in execution time. While GPUs generally operate at lower clock speeds compared to CPUs, their ability to handle large-scale parallel tasks offers a significant advantage.

For the ORd 2017 cell model, the GPU-based simulation achieved a speedup of up to 7.78 times compared to the 10-core CPU implementation.

[Figure 3. 6] Simulation time comparison between GPU and CPU in ORd 2017

As shown in figure 3.6, GPU parallelisation is not as dominant compared to GPU parallelisation in ORd 2011. This lower speedup compared to the ORd 2011 cell model is attributed to the use of the Forward Euler method, which is computationally more demanding than the Rush-Larsen method. Nevertheless, the GPU remains significantly more efficient for large-scale simulations.

## GPU Simulation Result Using ToR-ORd Cell Model

This section highlights the results obtained from GPU simulations using the ToR-ORd cell model. Similar to the ORd 2017 model, the forward Euler method was employed as the ODE solver. While this method provides adequate numerical stability and robustness for the ToR-ORD model, it results in longer computational times compared to the more efficient Rush-Larsen method used in the ORd 2011 simulations. Regardless of the computational time drawback, forward Euler method ensure the simulation generates a reliable and usable result.

### Result Validation

The validation process for the ToR-ORd cell model involved comparing GPU simulation outputs with the benchmark results obtained from OpenCOR. Time-series plots of action potentials were evaluated for qualitative consistency, and key biomarkers, such as action potential duration, ionic current profiles, and calcium dynamics, were quantitatively assessed under no-drug conditions.

[Figure 3. 7] Action Potential (mV) Shape of both CPU (dashed blue) and GPU (orange) Result Using ToR-ORd

The results from figure 3.7 confirmed the accuracy of the GPU simulation, as it produced outputs that matched the CPU-based OpenCOR simulations without any discernible differences. This validation underscores the reliability of the GPU implementation for the ToR-ORd cell model, even when employing the forward Euler method. Figure 3.6 illustrates the action potential traces from both GPU and CPU simulations, demonstrating their near-identical nature.

### Result Validation Under Drug

The validation of the GPU-based simulation for the ToR-ORd cell model under drug conditions was conducted using the same approach as for the ORd 2017 model. Drug effects were incorporated into the simulation by altering ionic currents based on predefined IC50 and Hill coefficient parameters. Both GPU and CPU (OpenCOR) simulations were configured identically to ensure fair comparison.

[Figure 3. 8] Action Potential (mV) Shape of both CPU (dashed) and GPU under drug effect Using ToR-ORd cell model

The results in figure 3.8 showed that the GPU simulation accurately replicated the outputs of the CPU-based simulations, with no significant differences observed in action potential traces or in biomarkers such as APD, ionic current dynamics, and calcium handling. This consistency demonstrates the validity of the GPU-based simulation for the ToR-ORd cell model, even when subjected to drug-induced perturbations. The successful validation of drug effects in the GPU simulations highlights the method's capability to simulate complex pharmacological scenarios reliably. These findings reinforce the utility of GPU-based simulations as a powerful tool for investigating drug-induced cellular behaviours.

### Computational Time and Efficiency Analysis

The computational performance for the ToR-ORd cell model was also analysed using the same hardware: an NVIDIA RTX 4090 GPU and a 10-core Intel Xeon (x86) Silver 4215 CPU @ 2.50 GHz. The GPU executed the simulations for 8000 samples (2000 samples per drug at four concentrations), matching the experimental conditions of the CPU. Similar to the ORd 2017 cell model, the computational time for the ToR-ORd model on CPUs increased linearly with the number of samples and pacing due to sequential processing. In contrast, the GPU’s architecture allowed for consistent computational performance, demonstrating the benefits of parallelisation for large-scale simulations.

[Figure 3. 9] Simulation time comparison between GPU and CPU in ToR-ORd cell model

As shown in figure 3.9, the ToR-ORd model GPU-based simulation achieved a speedup of up to 9.44 times compared to the 10-core CPU implementation. While not as high as the ORd 2011 model, this speedup highlights the GPU’s capability to handle complex simulations efficiently, even with computationally intensive solvers like Forward Euler. Overall, the GPU provides a substantial time-saving advantage across all tested cell models.

# Conclusion and Limitation

## Conclusion

This study effectively addressed the computational challenges in *in-silico* simulations for predicting cardiovascular drug toxicity by leveraging CUDA-based parallel processing. The optimized GPU approach achieved up to 40.91 times faster simulation speeds compared to traditional 10-core CPU methods, enabling the handling of larger datasets without significant performance loss. Validation results confirmed the accuracy of GPU simulations, even under suboptimal conditions, demonstrating their reliability for efficient and precise toxicity predictions. This advancement positions GPU-based methods as a cost-effective and practical alternative to CPU-based simulations in large-scale drug discovery research.

## Suggestions

To further enhance the utility and impact of this research, several suggestions can be made. Expanding the model complexity by incorporating additional biological variables could improve the accuracy of simulations. Testing the methodology on diverse GPU hardware models would provide insights into performance variations across systems, ensuring broader applicability. The method’s potential for industrial application is also notable, particularly in pharmaceutical pipelines where it could support large-scale drug development.

Furthermore, assessing the economic viability of the GPU-based approach, including long-term operational costs and energy efficiency, could ensure its sustainability. Standardising protocols for simulation workflows would help maintain consistency and reproducibility, which are critical for scientific and industrial adoption. Finally, fostering multidisciplinary collaboration with experts in pharmacology, bioinformatics, and hardware engineering could enhance the methodology's robustness and broaden its applicability. By addressing these areas, this CUDA-based approach could further solidify its role as a transformative tool in *in-silico* drug discovery, reducing reliance on animal testing and accelerating the pace of research.

# [References]

1. Jason Sanders and Edward Kandrot. 2010. CUDA by Example: An Introduction to General-Purpose GPU Programming (1st. ed.). Addison-Wesley Professional.

2. “What is OpenMP?” Accessed: Nov. 10, 2024. [Online]. Available: https://cvw.cac.cornell.edu/openmp/intro/what-is-openmp.

3. R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca and A. Lumsdaine, "Open MPI: A High-Performance, Heterogeneous MPI," 2006 IEEE International Conference on Cluster Computing, Barcelona, Spain, 2006, pp. 1-9, doi: 10.1109/CLUSTR.2006.311904.

4. A. Garny et al., “CellML and associated tools and techniques,” Sep. 13, 2008, Royal Society. doi: 10.1098/rsta.2008.0094.

5. M. Gómez, J. Carro, E. Pueyo, A. Pérez, A. Oliván, and V. Monasterio, “In Silico Modeling and Validation of the Effect of Calcium-Activated Potassium Current on Ventricular Repolarization in Failing Myocytes,” *IEEE J Biomed Health Inform*, pp. 1–9, 2024, doi: 10.1109/JBHI.2024.3495027.

6. C. M. Lloyd, J. R. Lawson, P. J. Hunter, and P. F. Nielsen, “The CellML Model Repository,” *Bioinformatics*, vol. 24, no. 18, pp. 2122–2123, 2008, doi: 10.1093/bioinformatics/btn390.

7. N. Le Novère et al., “BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems.,” Nucleic Acids Res, vol. 34, no. Database issue, 2006, doi: 10.1093/nar/gkj092.

8. M. Berghoff, J. Rosenbauer, F. Hoffmann, and A. Schug, “Cells in Silico-introducing a high-performance framework for large-scale tissue modeling,” *BMC Bioinformatics*, vol. 21, no. 1, Oct. 2020, doi: 10.1186/s12859-020-03728-7.

9. M. Martínez-del-Amor, I. Pérez-Hurtado, D. Orellana-Martín, and M. J. Pérez-Jiménez, “Adaptative parallel simulators for bioinspired computing models,” *Future Generation Computer Systems*, vol. 107, pp. 469–484, Jun. 2020, doi: 10.1016/j.future.2020.02.012.

10. S. McIntosh-Smith, J. Price, R. B. Sessions, and A. A. Ibarra, “High performance in silico virtual drug screening on many-core processors,” *International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 119–134, May 2015, doi: 10.1177/1094342014528252.

11. P. Amar, M. Baillieul, D. Barth, B. LeCun, F. Quessette, and S. Vial, “Parallel Biological In Silico Simulation,” Nov. 2014, doi: 10.1007/978-3-319-09465-6\_40.

12. D. G. Whittaker, J. C. Hancox, and H. Zhang, “*In Silico* Assestment of Pharmacotherapy for Human Atrial Patho-Electrophysiology Associated With hERG-Linked Short QT Syndrome” Jan. 2019, doi: 10.3389/fphys.2018.01888.

13. T. O'Hara, L. Virág, A. Varró, and Y. Rudy, “Simulation of the Undiseased Human Cardiac Ventricular Action Potential: Model Formulation and Experimental Validation.” 2011, PLOS Computational Biology 7(5): e1002061. Available: https://doi.org/10.1371/journal.pcbi.1002061.

14. S. Dutta, K.C. Chang, K.A. Beattie, J. Sheng , P.N. Tran, W.W. Wu, M. Wu, D.G. Strauss, T. Colatsky, and Z. Li. "Optimization of an In silico Cardiac Cell Model for Proarrhythmia Risk Assessment." Front Physiol. Aug 2017 doi: 10.3389/fphys.2017.00616.

15. Tomek, Jakub et al. “Development, calibration, and validation of a novel human ventricular myocyte model in health, disease, and drug block.” eLife vol. 8 e48890. Dec 2019, doi:10.7554/eLife.48890.

16. “Parallel Thread Execution ISA Version 8.5” Accessed: Nov. 24, 2024. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

17. C., Yves, C. D. Lontsi, and C. Pierre. "Rush-Larsen time-stepping methods of high order for stiff problems in cardiac electrophysiology." 2017, arXiv preprint arXiv:1712.02260.

# Appendix

## Project Structure

Codes for this research is available on Github repository of Computational Medicine Laboratory, Kumoh National Institute of Technology (<https://github.com/kit-cml>). There are some versions of the code in our Github profile, depeding on the cell models and research scenario. This section will discuss deeply the code’s version for ORd 2011 cell model (<https://github.com/kit-cml/MultiConcGPU>). Codes explained in this appendix is in same condition as the time this thesis being written, including some files that might be deleted due to redundancy in the future. More recent updates are available in the Github repository. Structure of the repository can be written as such:

MultiConcGPU

├─ .gitignore

├─ Makefile

├─ Makefile\_commercial

├─ bin

│  ├─ CVAR

│  │  ├─ HF\_male.csv

│  │  └─ sens\_healthy\_male.csv

│  ├─ autorun.sh

│  ├─ control

│  │  ├─ IC50\_samples.csv

│  │  └─ init\_state.csv

│  ├─ drug

│  │  ├─ IC50\_terfenadine.csv

│  │  ├─ IC50\_verapamil.csv

│  │  └─ testing

│  │     └─ IC50\_Mexiletine.csv

│  ├─ drug\_sim

│  ├─ ic50\_sens

│  │  └─ mitoxantrone\_sens.csv

│  ├─ input\_deck.txt

│  ├─ mitoxantrone

│  │  └─ mitoxantrone\_100\_samples\_50\_conc.csv

│  └─ result

│     ├─ do\_not\_delete\_this\_folder

│     └─ state\_only.zip

├─ cellmodels

│  ├─ Ohara\_Rudy\_2011.cpp

│  ├─ Ohara\_Rudy\_2011.hpp

│  ├─ cellmodel.hpp

│  └─ enums

│     ├─ enum\_Ohara\_Rudy\_2011.hpp

│     └─ enum\_ord2011.hpp

├─ main.cu

├─ modules

│  ├─ cipa\_t.cu

│  ├─ cipa\_t.cuh

│  ├─ drug\_conc.cpp

│  ├─ drug\_conc.hpp

│  ├─ glob\_funct.cpp

│  ├─ glob\_funct.hpp

│  ├─ glob\_type.cpp

│  ├─ glob\_type.hpp

│  ├─ gpu.cu

│  ├─ gpu.cuh

│  ├─ gpu\_cu\_.backup

│  ├─ gpu\_cuh.backup

│  ├─ gpu\_glob\_type.cu

│  ├─ gpu\_glob\_type.cuh

│  ├─ param.cpp

│  └─ param.hpp

└─ test\_compile.bat

The main structure, such as folder names, will less likely to be revised. Scripts inside the folder, especially in the ‘modules’ folder, more likely to be changed by removing some redundant functions. Next section will deeply discuss each file and their possibility of redundancy.

## Root folder

Root folder is the main folder that contains makefile, gitignore file, main code, and test\_compile.bat. The next sub section will disscuss each of them deeply.

## Makefile

This script is used for compiling the whole project in correct order, and enables easy cleanup and re-compilation. Binary files will be cleaned, and the simulator will be re-compiled with ‘make clean all’ command. Compilation configuration and steps described as follow:

# Some notes:

# - Using ':=' instead of '=' assign the value at Makefile parsing time,

# others are evaluated at usage time. This discards

# - Use ':set list' in Vi/Vim to show tabs (Ctrl-v-i force tab insertion)

#

# List to '.PHONY' all fake targets, those that are neither files nor folders.

# "all" and "clean" are good candidates.

.PHONY: all, clean

# Define the final program name

PROGNAME := drug\_sim

# Pre-processor flags to be used for includes (-I) and defines (-D)

CPPFLAGS := -I/usr/local/cuda/include

# CPPFLAGS :=

# CXX to set the compiler

# CXX := g++

CXX := nvcc

CXXLINK := nvlink

# CXXFLAGS is used for C++ compilation options.

#CXXFLAGS += -Wall -O0 -fpermissive -std=c++11

#CXXFLAGS += -Wall -O2 -fno-alias -fpermissive

# CXXFLAGS += -Wall

# Use this if you want to use ToR-ORd 2019 cell model.

# Otherwise, comment it

#CXXFLAGS += -DTOR-ORD\_2019

# LDFLAGS is used for linker (-g enables debug symbols)

# LDFLAGS += -g -L/usr/local/cuda/lib64

LDFLAGS += -g -L/usr/local/cuda/lib64 -arch=sm\_86 -rdc=true

# List the project' sources to compile or let the Makefile recognize

# them for you using 'wildcard' function.

#

SOURCES = $(wildcard \*.cpp) $(wildcard \*\*/\*.cpp) $(wildcard \*.c) $(wildcard \*\*/\*.c) $(wildcard \*\*/\*.cu) $(wildcard \*.cu)

# List the project' headers or let the Makefile recognize

# them for you using 'wildcard' function.

#

HEADERS = $(wildcard \*.hpp) $(wildcard \*\*/\*.hpp) $(wildcard \*.h) $(wildcard \*\*/\*.h) $(wildcard \*\*/\*.cuh) $(wildcard \*.cuh)

# Construct the list of object files based on source files using

# simple extension substitution.

OBJECTS := $(SOURCES:%.cpp=%.o)

LIBS= -lopenblas -lpthread -lcudart -lcublas

#

# Now declare the dependencies rules and targets

#

# Starting with 'all' make it becomes the default target when none

# is specified on 'make' command line.

all : $(PROGNAME)

# Declare that the final program depends on all objects and the Makfile

$(PROGNAME) : $(OBJECTS) Makefile

$(CXX) -o bin/$@ $(OBJECTS) $(LDFLAGS)

# Now the choice of using implicit rules or not (my choice)...

#

# Choice 1: use implicit rules and then we only need to add some dependencies

# to each object.

#

## Tells make that each object file depends on all headers and this Makefile.

#$(OBJECTS) : $(HEADERS) Makefile

#

# Choice 2: don't use implicit rules and specify our will

%.o: %.cpp $(HEADERS) Makefile

$(CXX) -x cu $(CXXFLAGS) $(CPPFLAGS) -dc -arch=sm\_86 $(OUTPUT\_OPTION) $<

# -dc -rdc=true

# Simple clean-up target

# notes:

# - the '@' before 'echo' informs make to hide command invocation.

# - the '-' before 'rm' command to informs make to ignore errors.

clean :

@echo "Clean."

rm -rf \*.o bin/$(PROGNAME)

rm -rf \*\*/\*.o

## .gitignore

.gitignore file specifies files and folders to be ignored by git, the version control used in this research. The git will ignore simulation results, binary files, CUDA related libraries, log files, and a jupyter notebook used in plotting. Below is the list of files and folder we ignore:

\*.i

\*.ii

\*.gpu

\*.ptx

\*.cubin

\*.fatbin

.DS\_Store

bin/drug\_sim

\*.o

\*.plt

\*.out

bin/result/\*/\*.csv

bin/result/\*/\*/\*.csv

bin/result/parse.ipynb

output.\*

\*.old

bin/result/\*

## main.cu

The main.cu file serves as the primary entry point for executing the drug simulation program. This file orchestrates the interaction between the core modules, manages the GPU-based computations, and handles input and output processes. Every CPU related orchestration for the simulation happen in main.cu. Its primary responsibilities are initialation, loading input data, initialise GPU environment, core number calculation, executing simulation, output handling, memory cleanup, logging, and making sure all input and output are correct. We found that CUDA debugger does not really help the debugging process due to unique parallelisation in this research. Self-created debugging points were also introduced in the main.cu. The main.cu coded as below:

#include <cuda.h>

#include <cuda\_runtime.h>

// #include "modules/drug\_sim.hpp"

#include "modules/glob\_funct.hpp"

#include "modules/glob\_type.hpp"

#include "modules/gpu.cuh"

#include "modules/cipa\_t.cuh"

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <iostream>

#include <math.h>

#include <vector>

#include <sys/stat.h>

#define ENOUGH ((CHAR\_BIT \* sizeof(int) - 1) / 3 + 2)

char buffer[255];

// unsigned int datapoint\_size = 7000;

const unsigned int sample\_limit = 10000;

clock\_t START\_TIMER;

clock\_t tic();

void toc(clock\_t start = START\_TIMER);

clock\_t tic()

{

return START\_TIMER = clock();

}

void toc(clock\_t start)

{

std::cout

<< "Elapsed time: "

<< (clock() - start) / (double)CLOCKS\_PER\_SEC << "s"

<< std::endl;

}

int gpu\_check(unsigned int datasize){

int num\_gpus;

float percent;

int id;

size\_t free, total;

cudaGetDeviceCount( &num\_gpus );

for ( int gpu\_id = 0; gpu\_id < num\_gpus; gpu\_id++ ) {

cudaSetDevice( gpu\_id );

cudaGetDevice( &id );

cudaMemGetInfo( &free, &total );

percent = (free/(float)total);

printf("GPU No %d\nFree Memory: %ld, Total Memory: %ld (%f percent free)\n", id,free,total,percent\*100.0);

}

percent = 1.0-(datasize/(float)total);

//// this code strangely gave out too small value, so i disable the safety switch for now

// printf("The program uses GPU No %d and %f percent of its memory\n", id,percent\*100.0);

// printf("\n");

// if (datasize<=free) {

// return 0;

// }

// else {

// return 1;

// }

return 0;

}

// get the IC50 data from file

drug\_t get\_IC50\_data\_from\_file(const char\* file\_name);

// return error and message based on the IC50 data

int check\_IC50\_content(const drug\_t\* ic50, const param\_t\* p\_param);

int get\_IC50\_data\_from\_file(const char\* file\_name, double \*ic50)

{

/\*

a host function to take all samples from the file, assuming each sample has 14 features.

it takes the file name, and an ic50 (already declared in 1D, everything become 1D)

as a note, the data will be stored in 1D array, means this functions applies flatten.

it returns 'how many samples were detected?' in integer.

\*/

FILE \*fp\_drugs;

// drug\_t ic50;

char \*token;

char buffer\_ic50[255];

unsigned int idx;

if( (fp\_drugs = fopen(file\_name, "r")) == NULL){

printf("Cannot open file %s\n",

file\_name);

return 0;

}

idx = 0;

int sample\_size = 0;

fgets(buffer\_ic50, sizeof(buffer\_ic50), fp\_drugs); // skip header

while( fgets(buffer\_ic50, sizeof(buffer\_ic50), fp\_drugs) != NULL )

{ // begin line reading

token = strtok( buffer\_ic50, "," );

while( token != NULL )

{ // begin data tokenizing

ic50[idx++] = strtod(token, NULL);

token = strtok(NULL, ",");

} // end data tokenizing

sample\_size++;

} // end line reading

fclose(fp\_drugs);

return sample\_size;

}

int get\_cvar\_data\_from\_file(const char\* file\_name, unsigned int limit, double \*cvar)

{

// buffer for writing in snprintf() function

char buffer\_cvar[255];

FILE \*fp\_cvar;

// cvar\_t cvar;

char \*token;

// std::array<double,18> temp\_array;

unsigned int idx;

if( (fp\_cvar = fopen(file\_name, "r")) == NULL){

printf("Cannot open file %s\n",

file\_name);

}

idx = 0;

int sample\_size = 0;

fgets(buffer\_cvar, sizeof(buffer\_cvar), fp\_cvar); // skip header

while( (fgets(buffer\_cvar, sizeof(buffer\_cvar), fp\_cvar) != NULL) && (sample\_size<limit))

{ // begin line reading

token = strtok( buffer\_cvar, "," );

while( token != NULL )

{ // begin data tokenizing

cvar[idx++] = strtod(token, NULL);

token = strtok(NULL, ",");

} // end data tokenizing

// printf("\n");

sample\_size++;

// cvar.push\_back(temp\_array);

} // end line reading

fclose(fp\_cvar);

return sample\_size;

}

int get\_init\_data\_from\_file(const char\* file\_name, double \*init\_states)

{

// buffer for writing in snprintf() function

char buffer\_cache[1023];

FILE \*fp\_cache;

// cvar\_t cvar;

char \*token;

// std::array<double,18> temp\_array;

unsigned long idx;

if( (fp\_cache = fopen(file\_name, "r")) == NULL){

printf("Cannot open file %s\n",

file\_name);

}

idx = 0;

unsigned int sample\_size = 0;

// fgets(buffer\_cvar, sizeof(buffer\_cvar), fp\_cvar); // skip header

while( (fgets(buffer\_cache, sizeof(buffer\_cache), fp\_cache) != NULL) )

{ // begin line reading

token = strtok( buffer\_cache, "," );

while( token != NULL )

{ // begin data tokenizing

init\_states[idx++] = strtod(token, NULL);

// if(idx < 82){

// printf("%d: %lf\n",idx-1,init\_states[idx-1]);

// }

token = strtok(NULL, ",");

} // end data tokenizing

// printf("\n");

sample\_size++;

// cvar.push\_back(temp\_array);

} // end line reading

fclose(fp\_cache);

return sample\_size;

}

int exists(const char \*fname)

{

FILE \*file;

if ((file = fopen(fname, "r")))

{

fclose(file);

return 1;

}

// fclose(file);

return 0;

}

int check\_IC50\_content(const drug\_t\* ic50, const param\_t\* p\_param)

{

if(ic50->size() == 0){

printf("Something problem with the IC50 file!\n");

return 1;

}

else if(ic50->size() > 2000){

printf( "Too much input! Maximum sample data is 2000!\n");

return 2;

}

else if(p\_param->pace\_max < 750 && p\_param->pace\_max > 1000){

printf("Make sure the maximum pace is around 750 to 1000!\n");

return 3;

}

// else if(mympi::size > ic50->size()){

// printf("%s\n%s\n",

// "Overflow of MPI Process!",

// "Make sure MPI Size is less than or equal the number of sample");

// return 4;

// }

else{

return 0;

}

}

int main(int argc, char \*\*argv)

{

// enable real-time output in stdout

//setvbuf( stdout, NULL, \_IONBF, 0 );

// NEW CODE STARTS HERE //

// mycuda \*thread\_id;

// cudaMalloc(&thread\_id, sizeof(mycuda));

This is how the main.cu manages multi-concentration in the simulation. Since ‘kernel\_DoDrugSim’ uses less memory, we can use them to put more samples, hence we can put more concentration in the modified IC50 file.

// TODO: Automation 3. check file inside folder

for (const auto &entry : fs::directory\_iterator(drug\_dir)) {

param\_t \*p\_param, \*d\_p\_param;

p\_param = new param\_t();

p\_param->init();

edison\_assign\_params(argc, argv, p\_param);

std::filesystem::directory\_entry dir\_entry = entry;

std::string entry\_str = dir\_entry.path().string();

std::cout << entry\_str << std::endl;

std::regex pattern("/([a-zA-Z0-9\_\.]+)\.csv");

std::smatch match;

std::regex\_search(entry\_str, match, pattern);

// TODO: Automation 2. create drug\_name and conc

// TODO: NewFile 2. disable drug name for now since the file name is inside it

// strcpy(p\_param->drug\_name, match[1].str().c\_str());

strcpy(p\_param->hill\_file, entry\_str.c\_str());

// strcat(p\_param->hill\_file, ".csv");

// strcat(p\_param->hill\_file, "/IC50\_samples.csv");

// TODO: NewFile 3. getvalue from source is unnecessary

// p\_param->conc = getValue(drugConcentration, match[1].str()) \* cmax;

// p\_param->show\_val();

// for qinwards calculation

double inal\_auc\_control = -90.547322; // AUC of INaL under control model

double ical\_auc\_control = -105.935067; // AUC of ICaL under control model

// input variables for cell simulation

param\_t \*p\_param = new param\_t(); // input data for CPU

param\_t \*d\_p\_param; // input data for GPU parsing

p\_param->init();

edison\_assign\_params(argc,argv,p\_param);

p\_param->show\_val();

double\* ic50 = (double \*)malloc(14 \* sample\_limit \* sizeof(double));

// if (p\_param->is\_cvar == true) cvar = (double \*)malloc(18 \* sample\_limit \* sizeof(double));

double\* cvar = (double \*)malloc(18 \* sample\_limit \* sizeof(double)); // conductance variability

const int num\_of\_constants = 146;

const int num\_of\_states = 41;

const int num\_of\_algebraic = 199;

const int num\_of\_rates = 41;

const double CONC = p\_param->conc;

Below is how the main.cu manages memory and data output for ‘kernel\_DoDrugSim\_single’. This kernel function requires more memory due to its more detailed output, hence requires a slightly adapted way to ensure all temporary memory wiped after simulation. In practice, this function also takes up to 60% more GPU memory compared to ‘kernel\_DoDrugSim’.

////////// if we are in write time series mode (post processing) //////////

if(p\_param->is\_time\_series == 1 /\*&& exists(p\_param->cache\_file) == 1 <- still unstable\*/){

printf("Using cached initial state from previous result!!!! \n\n");

const unsigned int datapoint\_size = p\_param->sampling\_limit; // sampling\_limit: limit of num of data points in one sample

double\* cache = (double \*)malloc((num\_of\_states+2) \* sample\_limit \* sizeof(double)); // array for in silico results

static const int CALCIUM\_SCALING = 1000000;

static const int CURRENT\_SCALING = 1000;

// snprintf(buffer, sizeof(buffer),

// "./drugs/bepridil/IC50\_samples.csv"

// // "./drugs/bepridil/IC50\_optimal.csv"

// // "./IC50\_samples.csv"

// );

int sample\_size = get\_IC50\_data\_from\_file(p\_param->hill\_file, ic50);

if(sample\_size == 0)

printf("Something problem with the IC50 file!\n");

// else if(sample\_size > 2000)

// printf("Too much input! Maximum sample data is 2000!\n");

printf("Sample size: %d\n",sample\_size);

printf("Set GPU Number: %d\n",p\_param->gpu\_index);

cudaSetDevice(p\_param->gpu\_index); // select a specific GPU

if(p\_param->is\_cvar == true){

int cvar\_sample = get\_cvar\_data\_from\_file(p\_param->cvar\_file,sample\_size,cvar);

printf("Reading: %d Conductance Variability samples\n",cvar\_sample);

}

printf("preparing GPU memory space \n");

// char buffer\_cvar[255];

// snprintf(buffer\_cvar, sizeof(buffer\_cvar),

// "./result/66\_00.csv"

// // "./drugs/optimized\_pop\_10k.csv"

// );

int cache\_num = get\_init\_data\_from\_file(p\_param->cache\_file,cache); //

printf("Found cache for %d samples\n",cache\_num);

// note to self:

// num of states+2 gave you at the very end of the file (pace number)

// the very beginning -> the core number

// for (int z = 0; z < num\_of\_states; z++) {printf("%lf\n", cache[z+1]);}

// printf("\n");

// for (int z = 0; z < num\_of\_states; z++) {printf("%lf\n", cache[ 1\*(num\_of\_states+2) + (z+2)]);}

// printf("\n");

// for (int z = 0; z < num\_of\_states; z++) {printf("%lf\n", cache[ 2\*(num\_of\_states+2) + (z+3)]);}

// return 0 ;

double \*d\_ic50;

double \*d\_cvar;

double \*d\_ALGEBRAIC;

double \*d\_CONSTANTS;

double \*d\_RATES;

double \*d\_STATES;

double \*d\_STATES\_cache;

// actually not used but for now, this is only for satisfiying the GPU regulator parameters

double \*d\_STATES\_RESULT;

double \*d\_all\_states;

cudaMalloc(&d\_ALGEBRAIC, num\_of\_algebraic \* sample\_size \* sizeof(double));

cudaMalloc(&d\_CONSTANTS, num\_of\_constants \* sample\_size \* sizeof(double));

cudaMalloc(&d\_RATES, num\_of\_rates \* sample\_size \* sizeof(double));

cudaMalloc(&d\_STATES, num\_of\_states \* sample\_size \* sizeof(double));

cudaMalloc(&d\_STATES\_cache, (num\_of\_states+2) \* sample\_size \* sizeof(double));

cudaMalloc(&d\_p\_param, sizeof(param\_t));

double \*time;

double \*dt;

double \*states;

double \*ical;

double \*inal;

double \*cai\_result;

double \*ina;

double \*ito;

double \*ikr;

double \*iks;

double \*ik1;

cipa\_t \*temp\_result, \*cipa\_result;

// prep for 1 cycle plus a bit (7000 \* sample\_size)

cudaMalloc(&temp\_result, sample\_size \* sizeof(cipa\_t)); // for temporal ??

cudaMalloc(&cipa\_result, sample\_size \* sizeof(cipa\_t)); // output of postprocessing

cudaMalloc(&time, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&dt, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&states, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&ical, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&inal, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&cai\_result, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&ina, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&ito, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&ikr, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&iks, sample\_size \* datapoint\_size \* sizeof(double));

cudaMalloc(&ik1, sample\_size \* datapoint\_size \* sizeof(double));

// cudaMalloc(&d\_STATES\_RESULT, (num\_of\_states+1) \* sample\_size \* sizeof(double));

// cudaMalloc(&d\_all\_states, num\_of\_states \* sample\_size \* p\_param->find\_steepest\_start \* sizeof(double));

cudaMalloc(&d\_ic50, sample\_size \* 14 \* sizeof(double)); // ic50s of 7 channels

cudaMalloc(&d\_cvar, sample\_size \* 18 \* sizeof(double)); // conductances of 18

printf("Copying sample files to GPU memory space \n");

cudaMemcpy(d\_STATES\_cache, cache, (num\_of\_states+2) \* sample\_size \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpy(d\_ic50, ic50, sample\_size \* 14 \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpy(d\_cvar, cvar, sample\_size \* 18 \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpy(d\_p\_param, p\_param, sizeof(param\_t), cudaMemcpyHostToDevice);

// // Get the maximum number of active blocks per multiprocessor

// cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, do\_drug\_sim\_analytical, threadsPerBlock);

// // Calculate the total number of blocks

// int numTotalBlocks = numBlocks \* cudaDeviceGetMultiprocessorCount();

tic();

printf("Timer started, doing simulation.... \n\n\nGPU Usage at this moment: \n");

int thread;

if (sample\_size>=16) thread = 16; // change this according to hardware

// optimal number of thread by experience -> might be different for each GPU, can be 16, can be 32

else thread = sample\_size;

// int block = int(ceil(sample\_size\*1.0/thread)+1);

int block = (sample\_size + thread - 1) / thread;

// int block = (sample\_size + thread - 1) / thread;

if(gpu\_check(15 \* sample\_size \* sizeof(double) + sizeof(param\_t)) == 1){

printf("GPU memory insufficient!\n");

return 0;

}

printf("Sample size: %d\n",sample\_size);

cudaSetDevice(p\_param->gpu\_index);

printf("\n Configuration: \n\n\tblock\t||\tthread\n---------------------------------------\n \t%d\t||\t%d\n\n\n", block,thread);

// initscr();

// printf("[\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] 0.00 %% \n");

kernel\_DrugSimulation<<<block,thread>>>(d\_ic50, d\_cvar, d\_CONSTANTS, d\_STATES, d\_STATES\_cache, d\_RATES, d\_ALGEBRAIC,

d\_STATES\_RESULT, d\_all\_states,

time, states, dt, cai\_result,

ina, inal,

ical, ito,

ikr, iks,

ik1,

sample\_size,

temp\_result, cipa\_result,

d\_p\_param

);

//block per grid, threads per block

// endwin();

cudaDeviceSynchronize();

printf("allocating memory for computation result in the CPU, malloc style \n");

double \*h\_states,\*h\_time,\*h\_dt,\*h\_ical,\*h\_inal,\*h\_cai\_result,\*h\_ina,\*h\_ito,\*h\_ikr,\*h\_iks,\*h\_ik1;

cipa\_t \*h\_cipa\_result;

h\_states = (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for STATES, \n");

h\_time = (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for time, \n");

h\_dt = (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for dt, \n");

h\_cai\_result= (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for Cai, \n");

h\_ina= (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for iNa, \n");

h\_ito= (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for ito, \n");

h\_ikr= (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for ikr, \n");

h\_iks= (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for iks, \n");

h\_ik1= (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for ik1, \n");

h\_ical= (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

printf("...allocated for ICaL, \n");

h\_inal = (double \*)malloc(datapoint\_size \* sample\_size \* sizeof(double));

h\_cipa\_result = (cipa\_t \*)malloc( sample\_size \* sizeof(cipa\_t));

printf("...allocating for INaL and postprocessing, all set!\n");

////// copy the data back to CPU, and write them into file ////////

printf("copying the data back to the CPU \n");

cudaMemcpy(h\_states, states, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_time, time, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_dt, dt, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_ical, ical, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_inal, inal, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_cai\_result, cai\_result, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_ina, ina, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_ito, ito, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_ikr, ikr, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_iks, iks, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_ik1, ik1, sample\_size \* datapoint\_size \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_cipa\_result, cipa\_result, sample\_size \* sizeof(cipa\_t), cudaMemcpyDeviceToHost);

FILE \*writer;

int check;

bool folder\_created = false;

printf("writing to file... \n");

// sample loop

for (int sample\_id = 0; sample\_id<sample\_size; sample\_id++){

// printf("writing sample %d... \n",sample\_id);

char sample\_str[ENOUGH];

char conc\_str[ENOUGH];

char filename[500] = "./result/";

sprintf(sample\_str, "%d", sample\_id);

sprintf(conc\_str, "%.2f", CONC);

strcat(filename,conc\_str);

strcat(filename,"/");

if (folder\_created == false){

check = mkdir(filename,0777);

// check if directory is created or not

if (!check){

printf("Directory created\n");

}

else {

printf("Unable to create directory, or the folder is already created, relax mate...\n");

}

folder\_created = true;

}

strcat(filename,sample\_str);

strcat(filename,"\_timeseries.csv");

writer = fopen(filename,"w");

fprintf(writer, "Time,Vm,dVm/dt,Cai,INa,INaL,ICaL,IKs,IKr,IK1,Ito\n");

for (int datapoint = 1; datapoint<datapoint\_size; datapoint++){

if (h\_time[ sample\_id + (datapoint \* sample\_size)] == 0.0) {break;}

fprintf(writer,"%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf\n", // change this into string, or limit the decimal accuracy, so we can decrease filesize

h\_time[ sample\_id + (datapoint \* sample\_size)],

h\_states[ sample\_id + (datapoint \* sample\_size)],

h\_dt[ sample\_id + (datapoint \* sample\_size)],

h\_cai\_result[ sample\_id + (datapoint \* sample\_size)],

h\_ina[ sample\_id + (datapoint \* sample\_size)],

h\_inal[ sample\_id + (datapoint \* sample\_size)],

h\_ical[ sample\_id + (datapoint \* sample\_size)],

h\_iks[ sample\_id + (datapoint \* sample\_size)],

h\_ikr[ sample\_id + (datapoint \* sample\_size)],

h\_ik1[ sample\_id + (datapoint \* sample\_size)],

h\_ito[ sample\_id + (datapoint \* sample\_size)]

);

}

fclose(writer);

}

printf("writing each biomarkers value... \n");

// sample loop

char conc\_str[ENOUGH];

char filename[500] = "./result/";

// sprintf(sample\_str, "%d", sample\_id);

sprintf(conc\_str, "%.2f", CONC);

strcat(filename,conc\_str);

strcat(filename,"/");

// printf("creating %s... \n", filename);

if (folder\_created == false){

check = mkdir(filename,0777);

// check if directory is created or not

if (!check){

printf("Directory created\n");

}

else {

printf("Unable to create directory, or the folder is already created, relax mate...\n");

}

folder\_created = true;

}

// strcat(filename,sample\_str);

strcat(filename,"\_biomarkers.csv");

writer = fopen(filename,"a");

fprintf(writer, "sample,qnet,qInward,inal\_auc,ical\_auc,apd90,apd50,apd\_tri,cad90,cad50,cad\_tri,dvmdt\_repol,vm\_peak,vm\_valley,vm\_dia,ca\_peak,ca\_valley,ca\_dia\n");

for (int sample\_id = 0; sample\_id<sample\_size; sample\_id++){

// printf("writing sample %d... \n",sample\_id);

fprintf(writer,"%d,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf\n", // change this into string, or limit the decimal accuracy, so we can decrease filesize

sample\_id,

h\_cipa\_result[sample\_id].qnet,

0.5\*((h\_cipa\_result[sample\_id].ical\_auc / ical\_auc\_control)+(h\_cipa\_result[sample\_id].inal\_auc / inal\_auc\_control)),

h\_cipa\_result[sample\_id].inal\_auc,

h\_cipa\_result[sample\_id].ical\_auc,

h\_cipa\_result[sample\_id].apd90,

h\_cipa\_result[sample\_id].apd50,

h\_cipa\_result[sample\_id].apd90 - h\_cipa\_result[sample\_id].apd50,

h\_cipa\_result[sample\_id].cad90,

h\_cipa\_result[sample\_id].cad50,

h\_cipa\_result[sample\_id].cad90 - h\_cipa\_result[sample\_id].cad50,

h\_cipa\_result[sample\_id].dvmdt\_repol,

h\_cipa\_result[sample\_id].vm\_peak,

h\_cipa\_result[sample\_id].vm\_valley,

h\_cipa\_result[sample\_id].vm\_dia,

h\_cipa\_result[sample\_id].ca\_peak,

h\_cipa\_result[sample\_id].ca\_valley,

h\_cipa\_result[sample\_id].ca\_dia

// temp\_result[sample\_id].qnet = 0.;

// temp\_result[sample\_id].inal\_auc = 0.;

// temp\_result[sample\_id].ical\_auc = 0.;

// temp\_result[sample\_id].dvmdt\_repol = -999;

// temp\_result[sample\_id].dvmdt\_max = -999;

// temp\_result[sample\_id].vm\_peak = -999;

// temp\_result[sample\_id].vm\_valley = d\_STATES[(sample\_id \* num\_of\_states) +V];

// temp\_result[sample\_id].vm\_dia = -999;

// temp\_result[sample\_id].apd90 = 0.;

// temp\_result[sample\_id].apd50 = 0.;

// temp\_result[sample\_id].ca\_peak = -999;

// temp\_result[sample\_id].ca\_valley = d\_STATES[(sample\_id \* num\_of\_states) +cai];

// temp\_result[sample\_id].ca\_dia = -999;

// temp\_result[sample\_id].cad90 = 0.;

// temp\_result[sample\_id].cad50 = 0.;

);

}

fclose(writer);

toc();

return 0;

}

The following is the *in silico* mode. This part of the main.cu explains how differ the handling for ‘kernel\_DoDrugSim’ compared to ‘kernel\_DoDrugSim\_single’. ‘kernel\_DoDrugSim’ requires less memory, so this following section of the code removes some memory assignments compared to ‘kernel\_DoDrugSim\_single:

////////// find cache mode (in silico code) //////////

else{

printf("In-silico mode, creating cache file because we don't have that yet, or is\_time\_series is intentionally false \n\n");

double \*d\_ic50;

double \*d\_cvar;

double \*d\_ALGEBRAIC;

double \*d\_CONSTANTS;

double \*d\_RATES;

double \*d\_STATES;

// not used, only to satisfy the parameters of the GPU regulator's function

double \*d\_STATES\_cache;

double \*time;

double \*dt;

double \*states;

double \*cai\_result;

double \*ical;

double \*inal;

double \*ina;

double \*ito;

double \*ikr;

double \*iks;

double \*ik1;

double \*d\_STATES\_RESULT;

double \*d\_all\_states;

cipa\_t \*temp\_result, \*cipa\_result;

// snprintf(buffer, sizeof(buffer),

// "./drugs/bepridil/IC50\_samples.csv"

// // "./drugs/bepridil/IC50\_optimal.csv"

// // "./IC50\_samples.csv"

// );

int sample\_size = get\_IC50\_data\_from\_file(p\_param->hill\_file, ic50);

if(sample\_size == 0)

printf("Something problem with the IC50 file!\n");

// else if(sample\_size > 2000)

// printf("Too much input! Maximum sample data is 2000!\n");

printf("Sample size: %d\n",sample\_size);

cudaSetDevice(p\_param->gpu\_index);

printf("preparing GPU memory space \n");

if(p\_param->is\_cvar == true){

int cvar\_sample = get\_cvar\_data\_from\_file(p\_param->cvar\_file,sample\_size,cvar);

printf("Reading: %d Conductance Variability samples\n",cvar\_sample);

}

cudaMalloc(&d\_ALGEBRAIC, num\_of\_algebraic \* sample\_size \* sizeof(double));

cudaMalloc(&d\_CONSTANTS, num\_of\_constants \* sample\_size \* sizeof(double));

cudaMalloc(&d\_RATES, num\_of\_rates \* sample\_size \* sizeof(double));

cudaMalloc(&d\_STATES, num\_of\_states \* sample\_size \* sizeof(double));

cudaMalloc(&d\_p\_param, sizeof(param\_t));

// prep for 1 cycle plus a bit (7000 \* sample\_size)

cudaMalloc(&temp\_result, sample\_size \* sizeof(cipa\_t));

cudaMalloc(&cipa\_result, sample\_size \* sizeof(cipa\_t));

cudaMalloc(&d\_STATES\_RESULT, (num\_of\_states+1) \* sample\_size \* sizeof(double)); // for cache file

cudaMalloc(&d\_all\_states, num\_of\_states \* sample\_size \* p\_param->find\_steepest\_start \* sizeof(double)); // for each sample

printf("Copying sample files to GPU memory space \n");

cudaMalloc(&d\_ic50, sample\_size \* 14 \* sizeof(double));

// if(p\_param->is\_cvar == true) cudaMalloc(&d\_cvar, sample\_size \* 18 \* sizeof(double));

cudaMalloc(&d\_cvar, sample\_size \* 18 \* sizeof(double));

cudaMemcpy(d\_ic50, ic50, sample\_size \* 14 \* sizeof(double), cudaMemcpyHostToDevice);

// if(p\_param->is\_cvar == true) cudaMemcpy(d\_cvar, cvar, sample\_size \* 18 \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpy(d\_cvar, cvar, sample\_size \* 18 \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpy(d\_p\_param, p\_param, sizeof(param\_t), cudaMemcpyHostToDevice);

// // Get the maximum number of active blocks per multiprocessor

// cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, do\_drug\_sim\_analytical, threadsPerBlock);

// // Calculate the total number of blocks

// int numTotalBlocks = numBlocks \* cudaDeviceGetMultiprocessorCount();

tic();

printf("Timer started, doing simulation.... \n GPU Usage at this moment: \n");

int thread;

if (sample\_size>=16){

thread = 16; // change this according to hardware

}

else thread = sample\_size;

// int block = int(ceil(sample\_size\*1.0/thread)+1);

int block = (sample\_size + thread - 1) / thread;

// int block = (sample\_size + thread - 1) / thread;

if(gpu\_check(15 \* sample\_size \* sizeof(double) + sizeof(param\_t)) == 1){

printf("GPU memory insufficient!\n");

return 0;

}

printf("Sample size: %d\n",sample\_size);

cudaSetDevice(p\_param->gpu\_index);

printf("\n Configuration: \n\n\tblock\t||\tthread\n---------------------------------------\n \t%d\t||\t%d\n\n\n", block,thread);

// initscr();

// printf("[\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_] 0.00 %% \n");

kernel\_DrugSimulation<<<block,thread>>>(d\_ic50, d\_cvar, d\_CONSTANTS, d\_STATES, d\_STATES\_cache, d\_RATES, d\_ALGEBRAIC,

d\_STATES\_RESULT, d\_all\_states,

time, states, dt, cai\_result,

ina, inal,

ical, ito,

ikr, iks,

ik1,

sample\_size,

temp\_result, cipa\_result,

d\_p\_param

);

//block per grid, threads per block

// endwin();

cudaDeviceSynchronize();

printf("allocating memory for computation result in the CPU, malloc style \n");

double \*h\_states, \*h\_all\_states;

cipa\_t \*h\_cipa\_result;

h\_states = (double \*)malloc((num\_of\_states+1) \* sample\_size \* sizeof(double)); //cache file

h\_all\_states = (double \*)malloc( (num\_of\_states) \* sample\_size \* p\_param->find\_steepest\_start \* sizeof(double)); //all core

h\_cipa\_result = (cipa\_t \*)malloc(sample\_size \* sizeof(cipa\_t));

printf("...allocating for all states, all set!\n");

////// copy the data back to CPU, and write them into file ////////

printf("copying the data back to the CPU \n");

cudaMemcpy(h\_cipa\_result, cipa\_result, sample\_size \* sizeof(cipa\_t), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_states, d\_STATES\_RESULT, sample\_size \* (num\_of\_states+1) \* sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_all\_states, d\_all\_states, (num\_of\_states) \* sample\_size \* p\_param->find\_steepest\_start \* sizeof(double), cudaMemcpyDeviceToHost);

FILE \*writer;

int check;

bool folder\_created = false;

// char sample\_str[ENOUGH];

char conc\_str[ENOUGH];

char filename[500] = "./result/";

sprintf(conc\_str, "%.2f", CONC);

strcat(filename,conc\_str);

// strcat(filename,"\_steepest");

if (folder\_created == false){

check = mkdir(filename,0777);

// check if directory is created or not

if (!check){

printf("Directory created\n");

}

else {

printf("Unable to create directory, or the folder is already created, relax mate...\n");

}

folder\_created = true;

}

// strcat(filename,sample\_str);

strcat(filename,".csv");

printf("writing to %s ... \n", filename);

writer = fopen(filename,"w");

// sample loop

for (int sample\_id = 0; sample\_id<sample\_size; sample\_id++){

// writer = fopen(filename,"a"); // because we have multiple fwrites

fprintf(writer,"%d,",sample\_id); // write core number at the front

for (int datapoint = 0; datapoint<num\_of\_states; datapoint++){

// if (h\_time[ sample\_id + (datapoint \* sample\_size)] == 0.0) {continue;}

fprintf(writer,"%.5f,", // change this into string, or limit the decimal accuracy, so we can decrease filesize

h\_states[(sample\_id \* (num\_of\_states+1)) + datapoint]

);

}

// fprintf(writer,"%lf,%lf\n", // write last data

// h\_states[(sample\_id \* num\_of\_states+1) + num\_of\_states],

// h\_states[(sample\_id \* num\_of\_states+1) + num\_of\_states+1]

// );

fprintf(writer,"%.5f\n", h\_states[(sample\_id \* (num\_of\_states+1))+num\_of\_states] );

// fprintf(writer, "\n");

// fclose(writer);

}

fclose(writer);

// // FILE \*writer;

// // int check;

// // bool folder\_created = false;

// printf("writing each core value... \n");

// // sample loop

// for (int sample\_id = 0; sample\_id<sample\_size; sample\_id++){

// // printf("writing sample %d... \n",sample\_id);

// char sample\_str[ENOUGH];

// char conc\_str[ENOUGH];

// char filename[500] = "./result/";

// sprintf(sample\_str, "%d", sample\_id);

// sprintf(conc\_str, "%.2f", CONC);

// strcat(filename,conc\_str);

// strcat(filename,"/");

// // printf("creating %s... \n", filename);

// if (folder\_created == false){

// check = mkdir(filename,0777);

// // check if directory is created or not

// if (!check){

// printf("Directory created\n");

// }

// else {

// printf("Unable to create directory, or the folder is already created, relax mate...\n");

// }

// folder\_created = true;

// }

// strcat(filename,sample\_str);

// strcat(filename,".csv");

// writer = fopen(filename,"w");

// for (int pacing = 0; pacing < p\_param->find\_steepest\_start; pacing++){ //pace loop

// // if (h\_time[ sample\_id + (datapoint \* sample\_size)] == 0.0) {continue;}

// for(int datapoint = 0; datapoint < num\_of\_states; datapoint++){ // each data loop

// fprintf(writer,"%lf,",h\_all\_states[((sample\_id \* num\_of\_states)) + ((sample\_size) \* pacing) + datapoint]);

// // fprintf(writer,"%lf,",h\_all\_states[((sample\_id \* num\_of\_states))+ datapoint]);

// }

// // fprintf(writer,"%d",p\_param->find\_steepest\_start + pacing);

// fprintf(writer,"%d\n",pacing + (p\_param->pace\_max - p\_param->find\_steepest\_start)+1 );

// }

// fclose(writer);

// }

toc();

return 0;

}

}

## test\_compile.bat

This file is a result from previous iteration. There was a moment that we wanted to develop the simulator for Windows Operating System outside of Linux. It is finally decided this repository will be containerised using Docker instead of separately develop Windows version of this simulator. This script will more likely to be deleted in the next update.

## ‘bin’ Folder

The bin folder serves as the central directory for storing various input files, intermediate data, and simulation outputs. This folder is organised into subdirectories that categorise data for ease of access and maintainability.

## CVAR

This folder will be used to store inter-individual conductance variability file for future development of this research.

## Control

The control subfolder contains files that are fundamental to running simulations under control conditions (without drug effects). It typically includes:

* IC50\_samples.csv: This file provides a baseline reference for the IC50 values of various ionic currents, which are used for comparison in simulations involving drug-induced conditions. IC50 file formatted as:

drug\_name,conc,ICaL\_IC50,ICaL\_h,IK1\_IC50,IK1\_h,IKs\_IC50,IKs\_h,INa\_IC50,INa\_h,INaL\_IC50,INaL\_h,Ito\_IC50,Ito\_h,hERG\_IC50,hERG\_h

bepridil,0,2704,0.6954,NA,NA,NA,NA,2371,1.984,1947,1.473,NA,NA,139.1,3.199

bepridil,0,2818,0.6409,NA,NA,NA,NA,2734,1.225,1802,1.212,NA,NA,181.4,2.77

bepridil,0,3939,0.718,NA,NA,NA,NA,3064,1.108,1921,1.421,NA,NA,194.8,0.8339

* init\_state.csv: This file contains the initial states of all variables in the cell model, such as membrane voltage and ion concentrations. These states serve as the starting point for the simulations, ensuring consistent and reproducible results.

By isolating control data in its dedicated subfolder, the simulation framework ensures clarity when comparing control and drug-altered conditions.

## drug

The drug subfolder stores input files related to simulations involving specific drugs. These files typically include the IC50 values and Hill coefficients for the drugs under study, which define their effects on various ionic currents. For example IC50\_terfenadine.csv and IC50\_verapamil.csv: These files describe the pharmacological properties of terfenadine and verapamil, respectively.

## result

The result subfolder is used to store simulation outputs, ensuring that data generated during the computational runs is systematically archived. Simulation output classified as two type, the init file, and post-processing folder. Init file is the output from the first phase of the simulation. It contains the simulation’s state when gradient of action potential is at its steepest. Named \_state\_only.csv in a folder named from the IC50 file, this initial state usually shaped like:

0,-89.14848,0.01218,0.00007,12.14017,12.14051,144.11277,144.11273,1.56242,1.55949,0.00008,0.00074,0.83611,0.83590,0.68309,0.83534,0.00015,0.53126,0.28205,0.00093,0.99964,0.56097,0.00047,0.99964,0.61777,0.00000,1.00000,0.92678,1.00000,0.99980,0.99996,1.00000,1.00000,0.00051,0.00087,0.00070,0.00083,0.99790,0.00002,0.00060,0.27721,0.00017,0.00000,0.00000,999.00000

1,-89.14444,0.01211,0.00007,12.12804,12.12838,144.10139,144.10135,1.55818,1.55521,0.00008,0.00074,0.83604,0.83582,0.68296,0.83526,0.00015,0.53107,0.28182,0.00093,0.99963,0.56062,0.00047,0.99963,0.61738,0.00000,1.00000,0.92684,1.00000,0.99980,0.99996,1.00000,1.00000,0.00051,0.00086,0.00070,0.00083,0.99789,0.00002,0.00060,0.27735,0.00017,0.00000,0.00000,999.00000

With the first column as sample ID number, and the last column acts as number of pace recorded. This file will become the input of the second phase. Post-processing folder contains a biomarker file and time-series data per sample. The biomarker file will be used for next researches that requires data analysis. We can validate the result, and manually analyse action potential and ion channels with the time-series files.

This subfolder's structure allows for easy retrieval of outputs for validation and analysis, ensuring that results from different runs are preserved without overwriting.

## ‘cellmodels’ Folder

This folder contains mathematical models of the cell. The folder is created to make cell models easier to change and modify. We put the converted C file in here, with an enum file as an addition.

## Ohara\_Rudy\_2011.hpp

This file declares all the function in ORd 2011 cell model. This header is required because the main parallelisation will be handled by gpu.cu, hence requiring an object-oriented programming approach. Notice that every function here is a kernel function, since each cell model will be simulated in parallel using each computing thread of the GPU. Below is the format of header file we implement, and this should be changed with any function changes in Ohara\_Rudy\_2011.cpp file:

#ifndef OHARA\_RUDY\_2011\_HPP

#define OHARA\_RUDY\_2011\_HPP

#include "enums/enum\_Ohara\_Rudy\_2011.hpp"

#include <cuda\_runtime.h>

// void initConsts();

// void initConsts(double type);

// void initConsts(bool is\_dutta);

\_\_device\_\_ void initConsts(double \*CONSTANTS, double \*STATES, double type, double conc, double \*ic50, double \*cvar, bool is\_dutta, bool is\_cvar, int offset);

\_\_device\_\_ void computeRates(double TIME, double\* CONSTANTS, double\* RATES, double\* STATES, double\* ALGEBRAIC, int offset);

\_\_device\_\_ void solveAnalytical(double \*CONSTANTS, double \*STATES, double \*ALGEBRAIC, double \*RATES, double dt, int offset);

\_\_device\_\_ void solveEuler( double \*STATES, double \*RATES, double dt, int offset);

\_\_device\_\_ double set\_time\_step(double TIME,double time\_point,double max\_time\_step,

double\* CONSTANTS,

double\* RATES,

double\* STATES,

double\* ALGEBRAIC,

int offset);

\_\_device\_\_ void applyDrugEffect(double \*CONSTANTS, double conc, double \*ic50, double epsilon, int offset);

\_\_device\_\_ void \_\_\_applyDutta(double \*CONSTANTS, int offset);

\_\_device\_\_ void \_\_\_applyCvar(double \*CONSTANTS, double \*cvar, int offset);

// void \_\_\_initConsts(double \*CONSTANTS, double \*STATES, double type, int offset);

#endif

## Ohara\_Rudy\_2011.cpp

This file contains the mathematical cell model and its solver. We added the solver and drug effect simulation function in this script. The rest of the script follows the conversion from CellML. Below is how we implement the modification and the whole script:

/\*

There are a total of 198 entries in the algebraic variable array.

There are a total of 41 entries in each of the rate and state variable arrays.

There are a total of 139+2 entries in the constant variable array.

\*/

#include "Ohara\_Rudy\_2011.hpp"

#include <cmath>

#include <cstdlib>

#include <cstdio>

#include "../modules/glob\_funct.hpp"

#include <cuda\_runtime.h>

#include <cuda.h>

\_\_device\_\_ void \_\_\_initConsts(double \*CONSTANTS, double \*STATES, double type, int offset)

{

int num\_of\_constants = 145;

int num\_of\_states = 41;

// printf("%d\n", offset);

CONSTANTS[(offset \* num\_of\_constants) + celltype] = type;

CONSTANTS[(offset \* num\_of\_constants) + nao] = 140;

CONSTANTS[(offset \* num\_of\_constants) + cao] = 1.8;

CONSTANTS[(offset \* num\_of\_constants) + ko] = 5.4;

CONSTANTS[(offset \* num\_of\_constants) + R] = 8314;

CONSTANTS[(offset \* num\_of\_constants) + T] = 310;

CONSTANTS[(offset \* num\_of\_constants) + F] = 96485;

CONSTANTS[(offset \* num\_of\_constants) + zna] = 1;

CONSTANTS[(offset \* num\_of\_constants) + zca] = 2;

CONSTANTS[(offset \* num\_of\_constants) + zk] = 1;

CONSTANTS[(offset \* num\_of\_constants) + L] = 0.01;

CONSTANTS[(offset \* num\_of\_constants) + rad] = 0.0011;

CONSTANTS[(offset \* num\_of\_constants) + stim\_start] = 10.0;

// bcl edited in the gpu.cu

CONSTANTS[(offset \* num\_of\_constants) + BCL] = 1000.0;

// cvar starts here

CONSTANTS[(offset \* num\_of\_constants) + Jrel\_scale] = 1.0;

CONSTANTS[(offset \* num\_of\_constants) + Jup\_scale] = 1.0;

CONSTANTS[(offset \* num\_of\_constants) + Jtr\_scale] = 1.0;

CONSTANTS[(offset \* num\_of\_constants) + Jleak\_scale] = 1.0;

//CONSTANTS[(offset \* num\_of\_constants) + KCaMK\_scale] = 1.0;

// cvar ends here

STATES[(offset \* num\_of\_states) + V] = -87;

CONSTANTS[(offset \* num\_of\_constants) + amp] = -80;

CONSTANTS[(offset \* num\_of\_constants) + duration] = 0.5;

CONSTANTS[(offset \* num\_of\_constants) + KmCaMK] = 0.15;

CONSTANTS[(offset \* num\_of\_constants) + aCaMK] = 0.05;

CONSTANTS[(offset \* num\_of\_constants) + bCaMK] = 0.00068;

CONSTANTS[(offset \* num\_of\_constants) + CaMKo] = 0.05;

CONSTANTS[(offset \* num\_of\_constants) + KmCaM] = 0.0015;

STATES[(offset \* num\_of\_states) + CaMKt] = 0;

STATES[(offset \* num\_of\_states) + cass] = 1e-4;

CONSTANTS[(offset \* num\_of\_constants) + cmdnmax\_b] = 0.05;

CONSTANTS[(offset \* num\_of\_constants) + kmcmdn] = 0.00238;

CONSTANTS[(offset \* num\_of\_constants) + trpnmax] = 0.07;

CONSTANTS[(offset \* num\_of\_constants) + kmtrpn] = 0.0005;

CONSTANTS[(offset \* num\_of\_constants) + BSRmax] = 0.047;

CONSTANTS[(offset \* num\_of\_constants) + KmBSR] = 0.00087;

CONSTANTS[(offset \* num\_of\_constants) + BSLmax] = 1.124;

CONSTANTS[(offset \* num\_of\_constants) + KmBSL] = 0.0087;

CONSTANTS[(offset \* num\_of\_constants) + csqnmax] = 10;

CONSTANTS[(offset \* num\_of\_constants) + kmcsqn] = 0.8;

STATES[(offset \* num\_of\_states) + nai] = 7;

STATES[(offset \* num\_of\_states) + nass] = 7;

STATES[(offset \* num\_of\_states) + ki] = 145;

STATES[(offset \* num\_of\_states) + kss] = 145;

STATES[(offset \* num\_of\_states) + cansr] = 1.2;

STATES[(offset \* num\_of\_states) + cajsr] = 1.2;

STATES[(offset \* num\_of\_states) + cai] = 1e-4;

CONSTANTS[(offset \* num\_of\_constants) + cm] = 1;

CONSTANTS[(offset \* num\_of\_constants) + PKNa] = 0.01833;

CONSTANTS[(offset \* num\_of\_constants) + mssV1] = 39.57;

CONSTANTS[(offset \* num\_of\_constants) + mssV2] = 9.871;

CONSTANTS[(offset \* num\_of\_constants) + mtV1] = 11.64;

CONSTANTS[(offset \* num\_of\_constants) + mtV2] = 34.77;

CONSTANTS[(offset \* num\_of\_constants) + mtD1] = 6.765;

CONSTANTS[(offset \* num\_of\_constants) + mtD2] = 8.552;

CONSTANTS[(offset \* num\_of\_constants) + mtV3] = 77.42;

CONSTANTS[(offset \* num\_of\_constants) + mtV4] = 5.955;

STATES[(offset \* num\_of\_states) + m] = 0;

CONSTANTS[(offset \* num\_of\_constants) + hssV1] = 82.9;

CONSTANTS[(offset \* num\_of\_constants) + hssV2] = 6.086;

CONSTANTS[(offset \* num\_of\_constants) + Ahf] = 0.99;

STATES[(offset \* num\_of\_states) + hf] = 1;

STATES[(offset \* num\_of\_states) + hs] = 1;

CONSTANTS[(offset \* num\_of\_constants) + GNa] = 75;

STATES[(offset \* num\_of\_states) + j] = 1;

STATES[(offset \* num\_of\_states) + hsp] = 1;

STATES[(offset \* num\_of\_states) + jp] = 1;

STATES[(offset \* num\_of\_states) + mL] = 0;

CONSTANTS[(offset \* num\_of\_constants) + thL] = 200;

STATES[(offset \* num\_of\_states) + hL] = 1;

STATES[(offset \* num\_of\_states) + hLp] = 1;

CONSTANTS[(offset \* num\_of\_constants) + GNaL\_b] = 0.0075;

CONSTANTS[(offset \* num\_of\_constants) + Gto\_b] = 0.02;

STATES[(offset \* num\_of\_states) + a] = 0;

STATES[(offset \* num\_of\_states) + iF] = 1;

STATES[(offset \* num\_of\_states) + iS] = 1;

STATES[(offset \* num\_of\_states) + ap] = 0;

STATES[(offset \* num\_of\_states) + iFp] = 1;

STATES[(offset \* num\_of\_states) + iSp] = 1;

CONSTANTS[(offset \* num\_of\_constants) + Kmn] = 0.002;

CONSTANTS[(offset \* num\_of\_constants) + k2n] = 1000;

CONSTANTS[(offset \* num\_of\_constants) + PCa\_b] = 0.0001;

STATES[(offset \* num\_of\_states) + d] = 0;

STATES[(offset \* num\_of\_states) + ff] = 1;

STATES[(offset \* num\_of\_states) + fs] = 1;

STATES[(offset \* num\_of\_states) + fcaf] = 1;

STATES[(offset \* num\_of\_states) + fcas] = 1;

STATES[(offset \* num\_of\_states) + jca] = 1;

STATES[(offset \* num\_of\_states) + ffp] = 1;

STATES[(offset \* num\_of\_states) + fcafp] = 1;

STATES[(offset \* num\_of\_states) + nca] = 0;

CONSTANTS[(offset \* num\_of\_constants) + GKr\_b] = 0.046;

STATES[(offset \* num\_of\_states) + xrf] = 0;

STATES[(offset \* num\_of\_states) + xrs] = 0;

CONSTANTS[(offset \* num\_of\_constants) + GKs\_b] = 0.0034;

STATES[(offset \* num\_of\_states) + xs1] = 0;

STATES[(offset \* num\_of\_states) + xs2] = 0;

CONSTANTS[(offset \* num\_of\_constants) + GK1\_b] = 0.1908;

STATES[(offset \* num\_of\_states) + xk1] = 1;

CONSTANTS[(offset \* num\_of\_constants) + kna1] = 15;

CONSTANTS[(offset \* num\_of\_constants) + kna2] = 5;

CONSTANTS[(offset \* num\_of\_constants) + kna3] = 88.12;

CONSTANTS[(offset \* num\_of\_constants) + kasymm] = 12.5;

CONSTANTS[(offset \* num\_of\_constants) + wna] = 6e4;

CONSTANTS[(offset \* num\_of\_constants) + wca] = 6e4;

CONSTANTS[(offset \* num\_of\_constants) + wnaca] = 5e3;

CONSTANTS[(offset \* num\_of\_constants) + kcaon] = 1.5e6;

CONSTANTS[(offset \* num\_of\_constants) + kcaoff] = 5e3;

CONSTANTS[(offset \* num\_of\_constants) + qna] = 0.5224;

CONSTANTS[(offset \* num\_of\_constants) + qca] = 0.167;

CONSTANTS[(offset \* num\_of\_constants) + KmCaAct] = 150e-6;

CONSTANTS[(offset \* num\_of\_constants) + Gncx\_b] = 0.0008;

CONSTANTS[(offset \* num\_of\_constants) + k1p] = 949.5;

CONSTANTS[(offset \* num\_of\_constants) + k1m] = 182.4;

CONSTANTS[(offset \* num\_of\_constants) + k2p] = 687.2;

CONSTANTS[(offset \* num\_of\_constants) + k2m] = 39.4;

CONSTANTS[(offset \* num\_of\_constants) + k3p] = 1899;

CONSTANTS[(offset \* num\_of\_constants) + k3m] = 79300;

CONSTANTS[(offset \* num\_of\_constants) + k4p] = 639;

CONSTANTS[(offset \* num\_of\_constants) + k4m] = 40;

CONSTANTS[(offset \* num\_of\_constants) + Knai0] = 9.073;

CONSTANTS[(offset \* num\_of\_constants) + Knao0] = 27.78;

CONSTANTS[(offset \* num\_of\_constants) + delta] = -0.155;

CONSTANTS[(offset \* num\_of\_constants) + Kki] = 0.5;

CONSTANTS[(offset \* num\_of\_constants) + Kko] = 0.3582;

CONSTANTS[(offset \* num\_of\_constants) + MgADP] = 0.05;

CONSTANTS[(offset \* num\_of\_constants) + MgATP] = 9.8;

CONSTANTS[(offset \* num\_of\_constants) + Kmgatp] = 1.698e-7;

CONSTANTS[(offset \* num\_of\_constants) + H] = 1e-7;

CONSTANTS[(offset \* num\_of\_constants) + eP] = 4.2;

CONSTANTS[(offset \* num\_of\_constants) + Khp] = 1.698e-7;

CONSTANTS[(offset \* num\_of\_constants) + Knap] = 224;

CONSTANTS[(offset \* num\_of\_constants) + Kxkur] = 292;

CONSTANTS[(offset \* num\_of\_constants) + Pnak\_b] = 30;

CONSTANTS[(offset \* num\_of\_constants) + GKb\_b] = 0.003;

CONSTANTS[(offset \* num\_of\_constants) + PNab] = 3.75e-10;

CONSTANTS[(offset \* num\_of\_constants) + PCab] = 2.5e-8;

CONSTANTS[(offset \* num\_of\_constants) + GpCa] = 0.0005;

CONSTANTS[(offset \* num\_of\_constants) + KmCap] = 0.0005;

CONSTANTS[(offset \* num\_of\_constants) + bt] = 4.75;

STATES[(offset \* num\_of\_states) + Jrelnp] = 0;

STATES[(offset \* num\_of\_states) + Jrelp] = 0;

CONSTANTS[(offset \* num\_of\_constants) + cmdnmax] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + cmdnmax\_b]\*1.30000 : CONSTANTS[(offset \* num\_of\_constants) + cmdnmax\_b]);

CONSTANTS[(offset \* num\_of\_constants) + Ahs] = 1.00000 - CONSTANTS[(offset \* num\_of\_constants) + Ahf];

CONSTANTS[(offset \* num\_of\_constants) + thLp] = 3.00000 \* CONSTANTS[(offset \* num\_of\_constants) + thL];

CONSTANTS[(offset \* num\_of\_constants) + GNaL] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GNaL\_b]\*0.600000 : CONSTANTS[(offset \* num\_of\_constants) + GNaL\_b]);

CONSTANTS[(offset \* num\_of\_constants) + Gto] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Gto\_b]\*4.00000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Gto\_b]\*4.00000 : CONSTANTS[(offset \* num\_of\_constants) + Gto\_b]);

CONSTANTS[(offset \* num\_of\_constants) + Aff] = 0.600000;

CONSTANTS[(offset \* num\_of\_constants) + PCa] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + PCa\_b]\*1.20000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + PCa\_b]\*2.50000 : CONSTANTS[(offset \* num\_of\_constants) + PCa\_b]);

CONSTANTS[(offset \* num\_of\_constants) + tjca] = 75.0000;

CONSTANTS[(offset \* num\_of\_constants) + GKr] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GKr\_b]\*1.30000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GKr\_b]\*0.800000 : CONSTANTS[(offset \* num\_of\_constants) + GKr\_b]);

CONSTANTS[(offset \* num\_of\_constants) + GKs] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GKs\_b]\*1.40000 : CONSTANTS[(offset \* num\_of\_constants) + GKs\_b]);

CONSTANTS[(offset \* num\_of\_constants) + GK1] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GK1\_b]\*1.20000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GK1\_b]\*1.30000 : CONSTANTS[(offset \* num\_of\_constants) + GK1\_b]);

CONSTANTS[(offset \* num\_of\_constants) + vcell] = 1000.00\*3.14000\*CONSTANTS[(offset \* num\_of\_constants) + rad]\*CONSTANTS[(offset \* num\_of\_constants) + rad]\*CONSTANTS[(offset \* num\_of\_constants) + L];

CONSTANTS[(offset \* num\_of\_constants) + GKb] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GKb\_b]\*0.600000 : CONSTANTS[(offset \* num\_of\_constants) + GKb\_b]);

CONSTANTS[(offset \* num\_of\_constants) + a\_rel] = 0.500000\*CONSTANTS[(offset \* num\_of\_constants) + bt];

CONSTANTS[(offset \* num\_of\_constants) + btp] = 1.25000\*CONSTANTS[(offset \* num\_of\_constants) + bt];

CONSTANTS[(offset \* num\_of\_constants) + upScale] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.0000 ? 1.30000 : 1.00000);

CONSTANTS[(offset \* num\_of\_constants) + Afs] = 1.00000 - CONSTANTS[(offset \* num\_of\_constants) + Aff];

CONSTANTS[(offset \* num\_of\_constants) + PCap] = 1.10000\*CONSTANTS[(offset \* num\_of\_constants) + PCa];

CONSTANTS[(offset \* num\_of\_constants) + PCaNa] = 0.00125000\*CONSTANTS[(offset \* num\_of\_constants) + PCa];

CONSTANTS[(offset \* num\_of\_constants) + PCaK] = 0.000357400\*CONSTANTS[(offset \* num\_of\_constants) + PCa];

CONSTANTS[(offset \* num\_of\_constants) + Ageo] = 2.00000\*3.14000\*CONSTANTS[(offset \* num\_of\_constants) + rad]\*CONSTANTS[(offset \* num\_of\_constants) + rad]+ 2.00000\*3.14000\*CONSTANTS[(offset \* num\_of\_constants) + rad]\*CONSTANTS[(offset \* num\_of\_constants) + L];

CONSTANTS[(offset \* num\_of\_constants) + a\_relp] = 0.500000\*CONSTANTS[(offset \* num\_of\_constants) + btp];

CONSTANTS[(offset \* num\_of\_constants) + PCaNap] = 0.00125000\*CONSTANTS[(offset \* num\_of\_constants) + PCap];

CONSTANTS[(offset \* num\_of\_constants) + PCaKp] = 0.000357400\*CONSTANTS[(offset \* num\_of\_constants) + PCap];

CONSTANTS[(offset \* num\_of\_constants) + Acap] = 2.00000\*CONSTANTS[(offset \* num\_of\_constants) + Ageo];

CONSTANTS[(offset \* num\_of\_constants) + vmyo] = 0.680000\*CONSTANTS[(offset \* num\_of\_constants) + vcell];

CONSTANTS[(offset \* num\_of\_constants) + vnsr] = 0.0552000\*CONSTANTS[(offset \* num\_of\_constants) + vcell];

CONSTANTS[(offset \* num\_of\_constants) + vjsr] = 0.00480000\*CONSTANTS[(offset \* num\_of\_constants) + vcell];

CONSTANTS[(offset \* num\_of\_constants) + vss] = 0.0200000\*CONSTANTS[(offset \* num\_of\_constants) + vcell];

CONSTANTS[(offset \* num\_of\_constants) + h10\_i] = CONSTANTS[(offset \* num\_of\_constants) + kasymm]+1.00000+ (CONSTANTS[(offset \* num\_of\_constants) + nao]/CONSTANTS[(offset \* num\_of\_constants) + kna1])\*(1.00000+CONSTANTS[(offset \* num\_of\_constants) + nao]/CONSTANTS[(offset \* num\_of\_constants) + kna2]);

CONSTANTS[(offset \* num\_of\_constants) + h11\_i] = ( CONSTANTS[(offset \* num\_of\_constants) + nao]\*CONSTANTS[(offset \* num\_of\_constants) + nao])/( CONSTANTS[(offset \* num\_of\_constants) + h10\_i]\*CONSTANTS[(offset \* num\_of\_constants) + kna1]\*CONSTANTS[(offset \* num\_of\_constants) + kna2]);

CONSTANTS[(offset \* num\_of\_constants) + h12\_i] = 1.00000/CONSTANTS[(offset \* num\_of\_constants) + h10\_i];

CONSTANTS[(offset \* num\_of\_constants) + k1\_i] = CONSTANTS[(offset \* num\_of\_constants) + h12\_i]\*CONSTANTS[(offset \* num\_of\_constants) + cao]\*CONSTANTS[(offset \* num\_of\_constants) + kcaon];

CONSTANTS[(offset \* num\_of\_constants) + k2\_i] = CONSTANTS[(offset \* num\_of\_constants) + kcaoff];

CONSTANTS[(offset \* num\_of\_constants) + k5\_i] = CONSTANTS[(offset \* num\_of\_constants) + kcaoff];

CONSTANTS[(offset \* num\_of\_constants) + Gncx] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Gncx\_b]\*1.10000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Gncx\_b]\*1.40000 : CONSTANTS[(offset \* num\_of\_constants) + Gncx\_b]);

CONSTANTS[(offset \* num\_of\_constants) + h10\_ss] = CONSTANTS[(offset \* num\_of\_constants) + kasymm]+1.00000+ (CONSTANTS[(offset \* num\_of\_constants) + nao]/CONSTANTS[(offset \* num\_of\_constants) + kna1])\*(1.00000+CONSTANTS[(offset \* num\_of\_constants) + nao]/CONSTANTS[(offset \* num\_of\_constants) + kna2]);

CONSTANTS[(offset \* num\_of\_constants) + h11\_ss] = ( CONSTANTS[(offset \* num\_of\_constants) + nao]\*CONSTANTS[(offset \* num\_of\_constants) + nao])/( CONSTANTS[(offset \* num\_of\_constants) + h10\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + kna1]\*CONSTANTS[(offset \* num\_of\_constants) + kna2]);

CONSTANTS[(offset \* num\_of\_constants) + h12\_ss] = 1.00000/CONSTANTS[(offset \* num\_of\_constants) + h10\_ss];

CONSTANTS[(offset \* num\_of\_constants) + k1\_ss] = CONSTANTS[(offset \* num\_of\_constants) + h12\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + cao]\*CONSTANTS[(offset \* num\_of\_constants) + kcaon];

CONSTANTS[(offset \* num\_of\_constants) + k2\_ss] = CONSTANTS[(offset \* num\_of\_constants) + kcaoff];

CONSTANTS[(offset \* num\_of\_constants) + k5\_ss] = CONSTANTS[(offset \* num\_of\_constants) + kcaoff];

CONSTANTS[(offset \* num\_of\_constants) + b1] = CONSTANTS[(offset \* num\_of\_constants) + k1m]\*CONSTANTS[(offset \* num\_of\_constants) + MgADP];

CONSTANTS[(offset \* num\_of\_constants) + a2] = CONSTANTS[(offset \* num\_of\_constants) + k2p];

CONSTANTS[(offset \* num\_of\_constants) + a4] = (( CONSTANTS[(offset \* num\_of\_constants) + k4p]\*CONSTANTS[(offset \* num\_of\_constants) + MgATP])/CONSTANTS[(offset \* num\_of\_constants) + Kmgatp])/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + MgATP]/CONSTANTS[(offset \* num\_of\_constants) + Kmgatp]);

CONSTANTS[(offset \* num\_of\_constants) + Pnak] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Pnak\_b]\*0.900000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Pnak\_b]\*0.700000 : CONSTANTS[(offset \* num\_of\_constants) + Pnak\_b]);

}

\_\_device\_\_ void \_\_\_applyDutta(double \*CONSTANTS, int offset)

{

int num\_of\_constants = 145;

//sisanya ganti jadi G (GKs for example)

CONSTANTS[GKs + (offset \* num\_of\_constants)] \*= 1.870;

CONSTANTS[GKr + (offset \* num\_of\_constants)] \*= 1.013;

CONSTANTS[GK1 + (offset \* num\_of\_constants)] \*= 1.698;

CONSTANTS[PCa + (offset \* num\_of\_constants)] \*= 1.007; //pca

CONSTANTS[GNaL + (offset \* num\_of\_constants)] \*= 2.661;

}

/\*==============\*/

/\* Added by ALI \*/

/\*==============\*/

\_\_device\_\_ void \_\_\_applyCvar(double \*CONSTANTS, double \*cvar, int offset)

{

int num\_of\_constants = 145;

CONSTANTS[(offset \* num\_of\_constants) +GNa] \*= cvar[0 + (offset\*18)]; // GNa

CONSTANTS[(offset \* num\_of\_constants) +GNaL] \*= cvar[1 + (offset\*18)]; // GNaL

CONSTANTS[(offset \* num\_of\_constants) +Gto] \*= cvar[2 + (offset\*18)]; // Gto

CONSTANTS[(offset \* num\_of\_constants) +GKr] \*= cvar[3 + (offset\*18)]; // GKr

CONSTANTS[(offset \* num\_of\_constants) +GKs] \*= cvar[4 + (offset\*18)]; // GKs

CONSTANTS[(offset \* num\_of\_constants) +GK1] \*= cvar[5 + (offset\*18)]; // GK1

CONSTANTS[(offset \* num\_of\_constants) +Gncx] \*= cvar[6 + (offset\*18)]; // GNaCa

CONSTANTS[(offset \* num\_of\_constants) +GKb] \*= cvar[7 + (offset\*18)]; // GKb

CONSTANTS[(offset \* num\_of\_constants) +PCa] \*= cvar[8 + (offset\*18)]; // PCa

CONSTANTS[(offset \* num\_of\_constants) +Pnak] \*= cvar[9 + (offset\*18)]; // INaK

CONSTANTS[(offset \* num\_of\_constants) +PNab] \*= cvar[10 + (offset\*18)]; // PNab

CONSTANTS[(offset \* num\_of\_constants) +PCab] \*= cvar[11 + (offset\*18)]; // PCab

CONSTANTS[(offset \* num\_of\_constants) +GpCa] \*= cvar[12 + (offset\*18)]; // GpCa

CONSTANTS[(offset \* num\_of\_constants) +KmCaMK] \*= cvar[17 + (offset\*18)]; // KCaMK

// Additional constants

CONSTANTS[(offset \* num\_of\_constants) +Jrel\_scale] \*= cvar[13 + (offset\*18)]; // SERCA\_Total (release)

CONSTANTS[(offset \* num\_of\_constants) +Jup\_scale] \*= cvar[14 + (offset\*18)]; // RyR\_Total (uptake)

CONSTANTS[(offset \* num\_of\_constants) +Jtr\_scale] \*= cvar[15 + (offset\*18)]; // Trans\_Total (NSR to JSR translocation)

CONSTANTS[(offset \* num\_of\_constants) +Jleak\_scale] \*= cvar[16 + (offset\*18)]; // Leak\_Total (Ca leak from NSR)

// CONSTANTS[(offset \* num\_of\_constants) +KCaMK\_scale] \*= cvar[17 + (offset\*18)]; // KCaMK

}

\_\_device\_\_ void applyDrugEffect(double \*CONSTANTS, double conc, double \*ic50, double epsilon, int offset)

{

int num\_of\_constants = 145;

CONSTANTS[PCa\_b+(offset \* num\_of\_constants)] = CONSTANTS[PCa\_b+(offset \* num\_of\_constants)] \* ( (ic50[0 + (offset\*14)] > epsilon && ic50[1+ (offset\*14)] > epsilon) ? 1./(1.+pow(conc/ic50[0+ (offset\*14)],ic50[1+ (offset\*14)])) : 1.);

CONSTANTS[GK1\_b+(offset \* num\_of\_constants)] = CONSTANTS[GK1\_b+(offset \* num\_of\_constants)] \* ((ic50[2 + (offset\*14)] > epsilon && ic50[3+ (offset\*14)] > epsilon) ? 1./(1.+pow(conc/ic50[2+ (offset\*14)],ic50[3+ (offset\*14)])) : 1.);

CONSTANTS[GKs\_b+(offset \* num\_of\_constants)] = CONSTANTS[GKs\_b+(offset \* num\_of\_constants)] \* ((ic50[4 + (offset\*14)] > epsilon && ic50[5+ (offset\*14)] > epsilon) ? 1./(1.+pow(conc/ic50[4+ (offset\*14)],ic50[5+ (offset\*14)])) : 1.);

CONSTANTS[GNa+(offset \* num\_of\_constants)] = CONSTANTS[GNa+(offset \* num\_of\_constants)] \* ((ic50[6 + (offset\*14)] > epsilon && ic50[7+ (offset\*14)] > epsilon) ? 1./(1.+pow(conc/ic50[6+ (offset\*14)],ic50[7+ (offset\*14)])) : 1.);

CONSTANTS[GNaL\_b+(offset \* num\_of\_constants)] = CONSTANTS[GNaL\_b+(offset \* num\_of\_constants)] \* ((ic50[8+ (offset\*14)] > epsilon && ic50[9+ (offset\*14)] > epsilon) ? 1./(1.+pow(conc/ic50[8+ (offset\*14)],ic50[9+ (offset\*14)])) : 1.);

CONSTANTS[Gto\_b+(offset \* num\_of\_constants)] = CONSTANTS[Gto\_b+(offset \* num\_of\_constants)] \* ((ic50[10 + (offset\*14)] > epsilon && ic50[11+ (offset\*14)] > epsilon) ? 1./(1.+pow(conc/ic50[10+ (offset\*14)],ic50[11+ (offset\*14)])) : 1.);

CONSTANTS[GKr\_b+(offset \* num\_of\_constants)] = CONSTANTS[GKr\_b+(offset \* num\_of\_constants)] \* ((ic50[12+ (offset\*14)] > epsilon && ic50[13+ (offset\*14)] > epsilon) ? 1./(1.+pow(conc/ic50[12+ (offset\*14)],ic50[13+ (offset\*14)])) : 1.);

}

// void initConsts(int offset)

// {

// \_\_\_initConsts(0.,offset);

// }

// void initConsts(double type)

// {

// \_\_\_initConsts(type, offset);

// }

\_\_device\_\_ void initConsts(double \*CONSTANTS, double \*STATES, double type, double conc, double \*ic50, double \*cvar, bool is\_dutta, bool is\_cvar, int offset)

{

// int num\_of\_constants = 145;

// printf("ic50:%d %lf, %lf, %lf\n",offset,ic50[0 + (offset\*14)],ic50[1 + (offset\*14)],ic50[2 + (offset\*14)]);

\_\_\_initConsts(CONSTANTS, STATES, type, offset); // initconst kan minta

// // mpi\_printf(0,"Celltype: %lf\n", CONSTANTS[celltype]);

// #ifndef COMPONENT\_PATCH // for patch clamp component based research

// // mpi\_printf(0,"Control %lf %lf %lf %lf %lf\n", CONSTANTS[PCa], CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);

// #endif

if(is\_dutta == true){

\_\_\_applyDutta(CONSTANTS, offset);

}

if(is\_cvar == true){

\_\_\_applyCvar(CONSTANTS, cvar, offset);

}

// #ifndef COMPONENT\_PATCH

// // mpi\_printf(0,"After Dutta %lf %lf %lf %lf %lf\n", CONSTANTS[PCa], CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);

// #endif

// \_\_\_applyDrugEffect(CONSTANTS, conc, ic50, 10E-14, offset);

// #ifndef COMPONENT\_PATCH

// // mpi\_printf(0,"After drug %lf %lf %lf %lf %lf\n", CONSTANTS[PCa], CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);

// #endif

}

\_\_device\_\_ void computeRates( double TIME, double \*CONSTANTS, double \*RATES, double \*STATES, double \*ALGEBRAIC, int offset )

{

int num\_of\_constants = 145; //done

int num\_of\_states = 41; //done

int num\_of\_algebraic = 199; //done

int num\_of\_rates = 41; //done

//new part

CONSTANTS[(offset \* num\_of\_constants) + cmdnmax] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + cmdnmax\_b]\*1.30000 : CONSTANTS[(offset \* num\_of\_constants) + cmdnmax\_b]);

CONSTANTS[(offset \* num\_of\_constants) + GNaL] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GNaL\_b]\*0.600000 : CONSTANTS[(offset \* num\_of\_constants) + GNaL\_b]);

CONSTANTS[(offset \* num\_of\_constants) + Gto] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Gto\_b]\*4.00000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Gto\_b]\*4.00000 : CONSTANTS[(offset \* num\_of\_constants) + Gto\_b]);

CONSTANTS[(offset \* num\_of\_constants) + PCa] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + PCa\_b]\*1.20000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + PCa\_b]\*2.50000 : CONSTANTS[(offset \* num\_of\_constants) + PCa\_b]);

CONSTANTS[(offset \* num\_of\_constants) + GKr] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GKr\_b]\*1.30000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GKr\_b]\*0.800000 : CONSTANTS[(offset \* num\_of\_constants) + GKr\_b]);

CONSTANTS[(offset \* num\_of\_constants) + GKs] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GKs\_b]\*1.40000 : CONSTANTS[(offset \* num\_of\_constants) + GKs\_b]);

CONSTANTS[(offset \* num\_of\_constants) + GK1] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GK1\_b]\*1.20000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GK1\_b]\*1.30000 : CONSTANTS[(offset \* num\_of\_constants) + GK1\_b]);

CONSTANTS[(offset \* num\_of\_constants) + GKb] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + GKb\_b]\*0.600000 : CONSTANTS[(offset \* num\_of\_constants) + GKb\_b]);

CONSTANTS[(offset \* num\_of\_constants) + upScale] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.0000 ? 1.30000 : 1.00000);

CONSTANTS[(offset \* num\_of\_constants) + Gncx] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Gncx\_b]\*1.10000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Gncx\_b]\*1.40000 : CONSTANTS[(offset \* num\_of\_constants) + Gncx\_b]);

CONSTANTS[(offset \* num\_of\_constants) + Pnak] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Pnak\_b]\*0.900000 : CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? CONSTANTS[(offset \* num\_of\_constants) + Pnak\_b]\*0.700000 : CONSTANTS[(offset \* num\_of\_constants) + Pnak\_b]);

// new part ends

ALGEBRAIC[(offset \* num\_of\_algebraic) +Istim] = (TIME>=CONSTANTS[(offset \* num\_of\_constants) + stim\_start] && (TIME - CONSTANTS[(offset \* num\_of\_constants) + stim\_start]) - floor((TIME - CONSTANTS[(offset \* num\_of\_constants) + stim\_start])/CONSTANTS[(offset \* num\_of\_constants) + BCL])\*CONSTANTS[(offset \* num\_of\_constants) + BCL]<=CONSTANTS[(offset \* num\_of\_constants) + duration] ? CONSTANTS[(offset \* num\_of\_constants) + amp] : 0.000000);

// in libcml there is ifdef TISSUE, ask further

ALGEBRAIC[(offset \* num\_of\_algebraic) +hLss] = 1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+87.6100)/7.48800));

ALGEBRAIC[(offset \* num\_of\_algebraic) +hLssp] = 1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+93.8100)/7.48800));

ALGEBRAIC[(offset \* num\_of\_algebraic) +mss] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V]+CONSTANTS[(offset \* num\_of\_constants) + mssV1])/CONSTANTS[(offset \* num\_of\_constants) + mssV2]));

ALGEBRAIC[(offset \* num\_of\_algebraic) +tm] = 1.00000/( CONSTANTS[(offset \* num\_of\_constants) + mtD1]\*exp((STATES[(offset \* num\_of\_states) + V]+CONSTANTS[(offset \* num\_of\_constants) + mtV1])/CONSTANTS[(offset \* num\_of\_constants) + mtV2])+ CONSTANTS[(offset \* num\_of\_constants) + mtD2]\*exp(- (STATES[(offset \* num\_of\_states) + V]+CONSTANTS[(offset \* num\_of\_constants) + mtV3])/CONSTANTS[(offset \* num\_of\_constants) + mtV4]));

ALGEBRAIC[(offset \* num\_of\_algebraic) +hss] = 1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+CONSTANTS[(offset \* num\_of\_constants) + hssV1])/CONSTANTS[(offset \* num\_of\_constants) + hssV2]));

ALGEBRAIC[(offset \* num\_of\_algebraic) +thf] = 1.00000/( 1.43200e-05\*exp(- (STATES[(offset \* num\_of\_states) + V]+1.19600)/6.28500)+ 6.14900\*exp((STATES[(offset \* num\_of\_states) + V]+0.509600)/20.2700));

ALGEBRAIC[(offset \* num\_of\_algebraic) +ths] = 1.00000/( 0.00979400\*exp(- (STATES[(offset \* num\_of\_states) + V]+17.9500)/28.0500)+ 0.334300\*exp((STATES[(offset \* num\_of\_states) + V]+5.73000)/56.6600));

ALGEBRAIC[(offset \* num\_of\_algebraic) +ass] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V] - 14.3400)/14.8200));

ALGEBRAIC[(offset \* num\_of\_algebraic) +ta] = 1.05150/(1.00000/( 1.20890\*(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V] - 18.4099)/29.3814)))+3.50000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+100.000)/29.3814)));

ALGEBRAIC[(offset \* num\_of\_algebraic) +dss] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V]+3.94000)/4.23000));

ALGEBRAIC[(offset \* num\_of\_algebraic) +td] = 0.600000+1.00000/(exp( - 0.0500000\*(STATES[(offset \* num\_of\_states) + V]+6.00000))+exp( 0.0900000\*(STATES[(offset \* num\_of\_states) + V]+14.0000)));

ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] = 1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+19.5800)/3.69600));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tff] = 7.00000+1.00000/( 0.00450000\*exp(- (STATES[(offset \* num\_of\_states) + V]+20.0000)/10.0000)+ 0.00450000\*exp((STATES[(offset \* num\_of\_states) + V]+20.0000)/10.0000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tfs] = 1000.00+1.00000/( 3.50000e-05\*exp(- (STATES[(offset \* num\_of\_states) + V]+5.00000)/4.00000)+ 3.50000e-05\*exp((STATES[(offset \* num\_of\_states) + V]+5.00000)/6.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] = ALGEBRAIC[(offset \* num\_of\_algebraic) + fss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + km2n] = STATES[(offset \* num\_of\_states) + jca]\*1.00000;

ALGEBRAIC[(offset \* num\_of\_algebraic) + anca] = 1.00000/(CONSTANTS[(offset \* num\_of\_constants) + k2n]/ALGEBRAIC[(offset \* num\_of\_algebraic) + km2n]+pow(1.00000+CONSTANTS[(offset \* num\_of\_constants) + Kmn]/STATES[(offset \* num\_of\_states) + cass], 4.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + xrss] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V]+8.33700)/6.78900));

ALGEBRAIC[(offset \* num\_of\_algebraic) + txrf] = 12.9800+1.00000/( 0.365200\*exp((STATES[(offset \* num\_of\_states) + V] - 31.6600)/3.86900)+ 4.12300e-05\*exp(- (STATES[(offset \* num\_of\_states) + V] - 47.7800)/20.3800));

ALGEBRAIC[(offset \* num\_of\_algebraic) + txrs] = 1.86500+1.00000/( 0.0662900\*exp((STATES[(offset \* num\_of\_states) + V] - 34.7000)/7.35500)+ 1.12800e-05\*exp(- (STATES[(offset \* num\_of\_states) + V] - 29.7400)/25.9400));

ALGEBRAIC[(offset \* num\_of\_algebraic) + xs1ss] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V]+11.6000)/8.93200));

ALGEBRAIC[(offset \* num\_of\_algebraic) + txs1] = 817.300+1.00000/( 0.000232600\*exp((STATES[(offset \* num\_of\_states) + V]+48.2800)/17.8000)+ 0.00129200\*exp(- (STATES[(offset \* num\_of\_states) + V]+210.000)/230.000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + xk1ss] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V]+ 2.55380\*CONSTANTS[(offset \* num\_of\_constants) + ko]+144.590)/( 1.56920\*CONSTANTS[(offset \* num\_of\_constants) + ko]+3.81150)));

ALGEBRAIC[(offset \* num\_of\_algebraic) + txk1] = 122.200/(exp(- (STATES[(offset \* num\_of\_states) + V]+127.200)/20.3600)+exp((STATES[(offset \* num\_of\_states) + V]+236.800)/69.3300));

ALGEBRAIC[(offset \* num\_of\_algebraic) + jss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + hss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + tj] = 2.03800+1.00000/( 0.0213600\*exp(- (STATES[(offset \* num\_of\_states) + V]+100.600)/8.28100)+ 0.305200\*exp((STATES[(offset \* num\_of\_states) + V]+0.994100)/38.4500));

ALGEBRAIC[(offset \* num\_of\_algebraic) + assp] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V] - 24.3400)/14.8200));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcaf] = 7.00000+1.00000/( 0.0400000\*exp(- (STATES[(offset \* num\_of\_states) + V] - 4.00000)/7.00000)+ 0.0400000\*exp((STATES[(offset \* num\_of\_states) + V] - 4.00000)/7.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcas] = 100.000+1.00000/( 0.000120000\*exp(- STATES[(offset \* num\_of\_states) + V]/3.00000)+ 0.000120000\*exp(STATES[(offset \* num\_of\_states) + V]/7.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tffp] = 2.50000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + tff];

ALGEBRAIC[(offset \* num\_of\_algebraic) + xs2ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + xs1ss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + txs2] = 1.00000/( 0.0100000\*exp((STATES[(offset \* num\_of\_states) + V] - 50.0000)/20.0000)+ 0.0193000\*exp(- (STATES[(offset \* num\_of\_states) + V]+66.5400)/31.0000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKb] = ( CONSTANTS[(offset \* num\_of\_constants) + CaMKo]\*(1.00000 - STATES[(offset \* num\_of\_states) + CaMKt]))/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + KmCaM]/STATES[(offset \* num\_of\_states) + cass]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + hssp] = 1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+89.1000)/6.08600));

ALGEBRAIC[(offset \* num\_of\_algebraic) + thsp] = 3.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + ths];

ALGEBRAIC[(offset \* num\_of\_algebraic) + tjp] = 1.46000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + tj];

ALGEBRAIC[(offset \* num\_of\_algebraic) + mLss] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V]+42.8500)/5.26400));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tmL] = ALGEBRAIC[(offset \* num\_of\_algebraic) + tm];

ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcafp] = 2.50000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcaf];

ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] = 1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+43.9400)/5.71100));

ALGEBRAIC[(offset \* num\_of\_algebraic) + delta\_epi] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==1.00000 ? 1.00000 - 0.950000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+70.0000)/5.00000)) : 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + tiF\_b] = 4.56200+1.00000/( 0.393300\*exp(- (STATES[(offset \* num\_of\_states) + V]+100.000)/100.000)+ 0.0800400\*exp((STATES[(offset \* num\_of\_states) + V]+50.0000)/16.5900));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tiF] = ALGEBRAIC[(offset \* num\_of\_algebraic) + tiF\_b]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + delta\_epi];

ALGEBRAIC[(offset \* num\_of\_algebraic) + tiS\_b] = 23.6200+1.00000/( 0.00141600\*exp(- (STATES[(offset \* num\_of\_states) + V]+96.5200)/59.0500)+ 1.78000e-08\*exp((STATES[(offset \* num\_of\_states) + V]+114.100)/8.07900));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tiS] = ALGEBRAIC[(offset \* num\_of\_algebraic) + tiS\_b]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + delta\_epi];

ALGEBRAIC[(offset \* num\_of\_algebraic) + dti\_develop] = 1.35400+0.000100000/(exp((STATES[(offset \* num\_of\_states) + V] - 167.400)/15.8900)+exp(- (STATES[(offset \* num\_of\_states) + V] - 12.2300)/0.215400));

ALGEBRAIC[(offset \* num\_of\_algebraic) + dti\_recover] = 1.00000 - 0.500000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+70.0000)/20.0000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + tiFp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + dti\_develop]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + dti\_recover]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + tiF];

ALGEBRAIC[(offset \* num\_of\_algebraic) + tiSp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + dti\_develop]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + dti\_recover]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + tiS];

ALGEBRAIC[(offset \* num\_of\_algebraic) + f] = CONSTANTS[(offset \* num\_of\_constants) + Aff]\*STATES[(offset \* num\_of\_states) + ff]+ CONSTANTS[(offset \* num\_of\_constants) + Afs]\*STATES[(offset \* num\_of\_states) + fs];

ALGEBRAIC[(offset \* num\_of\_algebraic) + Afcaf] = 0.300000+0.600000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V] - 10.0000)/10.0000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + Afcas] = 1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + Afcaf];

ALGEBRAIC[(offset \* num\_of\_algebraic) + fca] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Afcaf]\*STATES[(offset \* num\_of\_states) + fcaf]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + Afcas]\*STATES[(offset \* num\_of\_states) + fcas];

ALGEBRAIC[(offset \* num\_of\_algebraic) + fp] = CONSTANTS[(offset \* num\_of\_constants) + Aff]\*STATES[(offset \* num\_of\_states) + ffp]+ CONSTANTS[(offset \* num\_of\_constants) + Afs]\*STATES[(offset \* num\_of\_states) + fs];

ALGEBRAIC[(offset \* num\_of\_algebraic) + fcap] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Afcaf]\*STATES[(offset \* num\_of\_states) + fcafp]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + Afcas]\*STATES[(offset \* num\_of\_states) + fcas];

ALGEBRAIC[(offset \* num\_of\_algebraic) + vffrt] = ( STATES[(offset \* num\_of\_states) + V]\*CONSTANTS[(offset \* num\_of\_constants) + F]\*CONSTANTS[(offset \* num\_of\_constants) + F])/( CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt] = ( STATES[(offset \* num\_of\_states) + V]\*CONSTANTS[(offset \* num\_of\_constants) + F])/( CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaL] = ( 4.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vffrt]\*( STATES[(offset \* num\_of\_states) + cass]\*exp( 2.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 0.341000\*CONSTANTS[(offset \* num\_of\_constants) + cao]))/(exp( 2.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKa] = ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKb]+STATES[(offset \* num\_of\_states) + CaMKt];

ALGEBRAIC[(offset \* num\_of\_algebraic) + fICaLp] = 1.00000/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + KmCaMK]/ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKa]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaL] = (1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + fICaLp])\*CONSTANTS[(offset \* num\_of\_constants) + PCa]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaL]\*STATES[(offset \* num\_of\_states) + d]\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + f]\*(1.00000 - STATES[(offset \* num\_of\_states) + nca])+ STATES[(offset \* num\_of\_states) + jca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + fca]\*STATES[(offset \* num\_of\_states) + nca])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + fICaLp]\*CONSTANTS[(offset \* num\_of\_constants) + PCap]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaL]\*STATES[(offset \* num\_of\_states) + d]\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + fp]\*(1.00000 - STATES[(offset \* num\_of\_states) + nca])+ STATES[(offset \* num\_of\_states) + jca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + fcap]\*STATES[(offset \* num\_of\_states) + nca]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_inf\_temp] = ( CONSTANTS[(offset \* num\_of\_constants) + a\_rel]\*- ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaL])/(1.00000+ 1.00000\*pow(1.50000/STATES[(offset \* num\_of\_states) + cajsr], 8.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_inf] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_inf\_temp]\*1.70000 : ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_inf\_temp]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_rel\_temp] = CONSTANTS[(offset \* num\_of\_constants) + bt]/(1.00000+0.0123000/STATES[(offset \* num\_of\_states) + cajsr]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_rel] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_rel\_temp]<0.00100000 ? 0.00100000 : ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_rel\_temp]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_temp] = ( CONSTANTS[(offset \* num\_of\_constants) + a\_relp]\*- ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaL])/(1.00000+pow(1.50000/STATES[(offset \* num\_of\_states) + cajsr], 8.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_infp] = (CONSTANTS[(offset \* num\_of\_constants) + celltype]==2.00000 ? ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_temp]\*1.70000 : ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_temp]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_relp\_temp] = CONSTANTS[(offset \* num\_of\_constants) + btp]/(1.00000+0.0123000/STATES[(offset \* num\_of\_states) + cajsr]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_relp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_relp\_temp]<0.00100000 ? 0.00100000 : ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_relp\_temp]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + EK] = (( CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T])/CONSTANTS[(offset \* num\_of\_constants) + F])\*log(CONSTANTS[(offset \* num\_of\_constants) + ko]/STATES[(offset \* num\_of\_states) + ki]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + AiF] = 1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V] - 213.600)/151.200));

ALGEBRAIC[(offset \* num\_of\_algebraic) + AiS] = 1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + AiF];

ALGEBRAIC[(offset \* num\_of\_algebraic) + i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + AiF]\*STATES[(offset \* num\_of\_states) + iF]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + AiS]\*STATES[(offset \* num\_of\_states) + iS];

ALGEBRAIC[(offset \* num\_of\_algebraic) + ip] = ALGEBRAIC[(offset \* num\_of\_algebraic) + AiF]\*STATES[(offset \* num\_of\_states) + iFp]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + AiS]\*STATES[(offset \* num\_of\_states) + iSp];

ALGEBRAIC[(offset \* num\_of\_algebraic) + fItop] = 1.00000/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + KmCaMK]/ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKa]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Ito] = CONSTANTS[(offset \* num\_of\_constants) + Gto]\*(STATES[(offset \* num\_of\_states) + V] - ALGEBRAIC[(offset \* num\_of\_algebraic) + EK])\*( (1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + fItop])\*STATES[(offset \* num\_of\_states) + a]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + i]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + fItop]\*STATES[(offset \* num\_of\_states) + ap]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + ip]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Axrf] = 1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+54.8100)/38.2100));

ALGEBRAIC[(offset \* num\_of\_algebraic) + Axrs] = 1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + Axrf];

ALGEBRAIC[(offset \* num\_of\_algebraic) + xr] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Axrf]\*STATES[(offset \* num\_of\_states) + xrf]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + Axrs]\*STATES[(offset \* num\_of\_states) + xrs];

ALGEBRAIC[(offset \* num\_of\_algebraic) + rkr] = ( (1.00000/(1.00000+exp((STATES[(offset \* num\_of\_states) + V]+55.0000)/75.0000)))\*1.00000)/(1.00000+exp((STATES[(offset \* num\_of\_states) + V] - 10.0000)/30.0000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + IKr] = CONSTANTS[(offset \* num\_of\_constants) + GKr]\* pow((CONSTANTS[(offset \* num\_of\_constants) + ko]/5.40000), 1.0 / 2)\*ALGEBRAIC[(offset \* num\_of\_algebraic) + xr]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + rkr]\*(STATES[(offset \* num\_of\_states) + V] - ALGEBRAIC[(offset \* num\_of\_algebraic) + EK]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + EKs] = (( CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T])/CONSTANTS[(offset \* num\_of\_constants) + F])\*log((CONSTANTS[(offset \* num\_of\_constants) + ko]+ CONSTANTS[(offset \* num\_of\_constants) + PKNa]\*CONSTANTS[(offset \* num\_of\_constants) + nao])/(STATES[(offset \* num\_of\_states) + ki]+ CONSTANTS[(offset \* num\_of\_constants) + PKNa]\*STATES[(offset \* num\_of\_states) + nai]));

ALGEBRAIC[(offset \* num\_of\_algebraic) + KsCa] = 1.00000+0.600000/(1.00000+pow(3.80000e-05/STATES[(offset \* num\_of\_states) + cai], 1.40000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + IKs] = CONSTANTS[(offset \* num\_of\_constants) + GKs]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + KsCa]\*STATES[(offset \* num\_of\_states) + xs1]\*STATES[(offset \* num\_of\_states) + xs2]\*(STATES[(offset \* num\_of\_states) + V] - ALGEBRAIC[(offset \* num\_of\_algebraic) + EKs]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + rk1] = 1.00000/(1.00000+exp(((STATES[(offset \* num\_of\_states) + V]+105.800) - 2.60000\*CONSTANTS[(offset \* num\_of\_constants) + ko])/9.49300));

ALGEBRAIC[(offset \* num\_of\_algebraic) + IK1] = CONSTANTS[(offset \* num\_of\_constants) + GK1]\* pow(CONSTANTS[(offset \* num\_of\_constants) + ko], 1.0 / 2)\*ALGEBRAIC[(offset \* num\_of\_algebraic) + rk1]\*STATES[(offset \* num\_of\_states) + xk1]\*(STATES[(offset \* num\_of\_states) + V] - ALGEBRAIC[(offset \* num\_of\_algebraic) + EK]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Knao] = CONSTANTS[(offset \* num\_of\_constants) + Knao0]\*exp(( (1.00000 - CONSTANTS[(offset \* num\_of\_constants) + delta])\*STATES[(offset \* num\_of\_states) + V]\*CONSTANTS[(offset \* num\_of\_constants) + F])/( 3.00000\*CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T]));

ALGEBRAIC[(offset \* num\_of\_algebraic) + a3] = ( CONSTANTS[(offset \* num\_of\_constants) + k3p]\*pow(CONSTANTS[(offset \* num\_of\_constants) + ko]/CONSTANTS[(offset \* num\_of\_constants) + Kko], 2.00000))/((pow(1.00000+CONSTANTS[(offset \* num\_of\_constants) + nao]/ALGEBRAIC[(offset \* num\_of\_algebraic) + Knao], 3.00000)+pow(1.00000+CONSTANTS[(offset \* num\_of\_constants) + ko]/CONSTANTS[(offset \* num\_of\_constants) + Kko], 2.00000)) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + P] = CONSTANTS[(offset \* num\_of\_constants) + eP]/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + H]/CONSTANTS[(offset \* num\_of\_constants) + Khp]+STATES[(offset \* num\_of\_states) + nai]/CONSTANTS[(offset \* num\_of\_constants) + Knap]+STATES[(offset \* num\_of\_states) + ki]/CONSTANTS[(offset \* num\_of\_constants) + Kxkur]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + b3] = ( CONSTANTS[(offset \* num\_of\_constants) + k3m]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + P]\*CONSTANTS[(offset \* num\_of\_constants) + H])/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + MgATP]/CONSTANTS[(offset \* num\_of\_constants) + Kmgatp]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Knai] = CONSTANTS[(offset \* num\_of\_constants) + Knai0]\*exp(( CONSTANTS[(offset \* num\_of\_constants) + delta]\*STATES[(offset \* num\_of\_states) + V]\*CONSTANTS[(offset \* num\_of\_constants) + F])/( 3.00000\*CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T]));

ALGEBRAIC[(offset \* num\_of\_algebraic) + a1] = ( CONSTANTS[(offset \* num\_of\_constants) + k1p]\*pow(STATES[(offset \* num\_of\_states) + nai]/ALGEBRAIC[(offset \* num\_of\_algebraic) + Knai], 3.00000))/((pow(1.00000+STATES[(offset \* num\_of\_states) + nai]/ALGEBRAIC[(offset \* num\_of\_algebraic) + Knai], 3.00000)+pow(1.00000+STATES[(offset \* num\_of\_states) + ki]/CONSTANTS[(offset \* num\_of\_constants) + Kki], 2.00000)) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + b2] = ( CONSTANTS[(offset \* num\_of\_constants) + k2m]\*pow(CONSTANTS[(offset \* num\_of\_constants) + nao]/ALGEBRAIC[(offset \* num\_of\_algebraic) + Knao], 3.00000))/((pow(1.00000+CONSTANTS[(offset \* num\_of\_constants) + nao]/ALGEBRAIC[(offset \* num\_of\_algebraic) + Knao], 3.00000)+pow(1.00000+CONSTANTS[(offset \* num\_of\_constants) + ko]/CONSTANTS[(offset \* num\_of\_constants) + Kko], 2.00000)) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + b4] = ( CONSTANTS[(offset \* num\_of\_constants) + k4m]\*pow(STATES[(offset \* num\_of\_states) + ki]/CONSTANTS[(offset \* num\_of\_constants) + Kki], 2.00000))/((pow(1.00000+STATES[(offset \* num\_of\_states) + nai]/ALGEBRAIC[(offset \* num\_of\_algebraic) + Knai], 3.00000)+pow(1.00000+STATES[(offset \* num\_of\_states) + ki]/CONSTANTS[(offset \* num\_of\_constants) + Kki], 2.00000)) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + x1] = CONSTANTS[(offset \* num\_of\_constants) + a4]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a1]\*CONSTANTS[(offset \* num\_of\_constants) + a2]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + b2]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b4]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b3]+ CONSTANTS[(offset \* num\_of\_constants) + a2]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b4]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b3]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + b3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a1]\*CONSTANTS[(offset \* num\_of\_constants) + a2];

ALGEBRAIC[(offset \* num\_of\_algebraic) + x2] = ALGEBRAIC[(offset \* num\_of\_algebraic) + b2]\*CONSTANTS[(offset \* num\_of\_constants) + b1]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b4]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + a1]\*CONSTANTS[(offset \* num\_of\_constants) + a2]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a3]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + a3]\*CONSTANTS[(offset \* num\_of\_constants) + b1]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b4]+ CONSTANTS[(offset \* num\_of\_constants) + a2]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b4];

ALGEBRAIC[(offset \* num\_of\_algebraic) + x3] = CONSTANTS[(offset \* num\_of\_constants) + a2]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a3]\*CONSTANTS[(offset \* num\_of\_constants) + a4]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + b3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b2]\*CONSTANTS[(offset \* num\_of\_constants) + b1]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + b2]\*CONSTANTS[(offset \* num\_of\_constants) + b1]\*CONSTANTS[(offset \* num\_of\_constants) + a4]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + a3]\*CONSTANTS[(offset \* num\_of\_constants) + a4]\*CONSTANTS[(offset \* num\_of\_constants) + b1];

ALGEBRAIC[(offset \* num\_of\_algebraic) + x4] = ALGEBRAIC[(offset \* num\_of\_algebraic) + b4]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b2]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + a3]\*CONSTANTS[(offset \* num\_of\_constants) + a4]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a1]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + b2]\*CONSTANTS[(offset \* num\_of\_constants) + a4]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a1]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + b3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b2]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a1];

ALGEBRAIC[(offset \* num\_of\_algebraic) + E1] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x1]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E2] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x2]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + JnakNa] = 3.00000\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + E1]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a3] - ALGEBRAIC[(offset \* num\_of\_algebraic) + E2]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + b3]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E3] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x3]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E4] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x4]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + JnakK] = 2.00000\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + E4]\*CONSTANTS[(offset \* num\_of\_constants) + b1] - ALGEBRAIC[(offset \* num\_of\_algebraic) + E3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + a1]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + INaK] = CONSTANTS[(offset \* num\_of\_constants) + Pnak]\*( CONSTANTS[(offset \* num\_of\_constants) + zna]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + JnakNa]+ CONSTANTS[(offset \* num\_of\_constants) + zk]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + JnakK]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + xkb] = 1.00000/(1.00000+exp(- (STATES[(offset \* num\_of\_states) + V] - 14.4800)/18.3400));

ALGEBRAIC[(offset \* num\_of\_algebraic) + IKb] = CONSTANTS[(offset \* num\_of\_constants) + GKb]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + xkb]\*(STATES[(offset \* num\_of\_states) + V] - ALGEBRAIC[(offset \* num\_of\_algebraic) + EK]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + JdiffK] = (STATES[(offset \* num\_of\_states) + kss] - STATES[(offset \* num\_of\_states) + ki])/2.00000;

ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaK] = ( 1.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vffrt]\*( 0.750000\*STATES[(offset \* num\_of\_states) + kss]\*exp( 1.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 0.750000\*CONSTANTS[(offset \* num\_of\_constants) + ko]))/(exp( 1.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaK] = (1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + fICaLp])\*CONSTANTS[(offset \* num\_of\_constants) + PCaK]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaK]\*STATES[(offset \* num\_of\_states) + d]\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + f]\*(1.00000 - STATES[(offset \* num\_of\_states) + nca])+ STATES[(offset \* num\_of\_states) + jca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + fca]\*STATES[(offset \* num\_of\_states) + nca])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + fICaLp]\*CONSTANTS[(offset \* num\_of\_constants) + PCaKp]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaK]\*STATES[(offset \* num\_of\_states) + d]\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + fp]\*(1.00000 - STATES[(offset \* num\_of\_states) + nca])+ STATES[(offset \* num\_of\_states) + jca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + fcap]\*STATES[(offset \* num\_of\_states) + nca]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + ENa] = (( CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T])/CONSTANTS[(offset \* num\_of\_constants) + F])\*log(CONSTANTS[(offset \* num\_of\_constants) + nao]/STATES[(offset \* num\_of\_states) + nai]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + h] = CONSTANTS[(offset \* num\_of\_constants) + Ahf]\*STATES[(offset \* num\_of\_states) + hf]+ CONSTANTS[(offset \* num\_of\_constants) + Ahs]\*STATES[(offset \* num\_of\_states) + hs];

ALGEBRAIC[(offset \* num\_of\_algebraic) + hp] = CONSTANTS[(offset \* num\_of\_constants) + Ahf]\*STATES[(offset \* num\_of\_states) + hf]+ CONSTANTS[(offset \* num\_of\_constants) + Ahs]\*STATES[(offset \* num\_of\_states) + hsp];

ALGEBRAIC[(offset \* num\_of\_algebraic) + fINap] = 1.00000/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + KmCaMK]/ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKa]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + INa] = CONSTANTS[(offset \* num\_of\_constants) + GNa]\*(STATES[(offset \* num\_of\_states) + V] - ALGEBRAIC[(offset \* num\_of\_algebraic) + ENa])\*pow(STATES[(offset \* num\_of\_states) + m], 3.00000)\*( (1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + fINap])\*ALGEBRAIC[(offset \* num\_of\_algebraic) + h]\*STATES[(offset \* num\_of\_states) + j]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + fINap]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + hp]\*STATES[(offset \* num\_of\_states) + jp]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + fINaLp] = 1.00000/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + KmCaMK]/ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKa]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + INaL] = CONSTANTS[(offset \* num\_of\_constants) + GNaL]\*(STATES[(offset \* num\_of\_states) + V] - ALGEBRAIC[(offset \* num\_of\_algebraic) + ENa])\*STATES[(offset \* num\_of\_states) + mL]\*( (1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + fINaLp])\*STATES[(offset \* num\_of\_states) + hL]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + fINaLp]\*STATES[(offset \* num\_of\_states) + hLp]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + allo\_i] = 1.00000/(1.00000+pow(CONSTANTS[(offset \* num\_of\_constants) + KmCaAct]/STATES[(offset \* num\_of\_states) + cai], 2.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + hna] = exp(( CONSTANTS[(offset \* num\_of\_constants) + qna]\*STATES[(offset \* num\_of\_states) + V]\*CONSTANTS[(offset \* num\_of\_constants) + F])/( CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T]));

ALGEBRAIC[(offset \* num\_of\_algebraic) + h7\_i] = 1.00000+ (CONSTANTS[(offset \* num\_of\_constants) + nao]/CONSTANTS[(offset \* num\_of\_constants) + kna3])\*(1.00000+1.00000/ALGEBRAIC[(offset \* num\_of\_algebraic) + hna]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + h8\_i] = CONSTANTS[(offset \* num\_of\_constants) + nao]/( CONSTANTS[(offset \* num\_of\_constants) + kna3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + hna]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + h7\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + k3pp\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h8\_i]\*CONSTANTS[(offset \* num\_of\_constants) + wnaca];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h1\_i] = 1.00000+ (STATES[(offset \* num\_of\_states) + nai]/CONSTANTS[(offset \* num\_of\_constants) + kna3])\*(1.00000+ALGEBRAIC[(offset \* num\_of\_algebraic) + hna]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + h2\_i] = ( STATES[(offset \* num\_of\_states) + nai]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + hna])/( CONSTANTS[(offset \* num\_of\_constants) + kna3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + h1\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + k4pp\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h2\_i]\*CONSTANTS[(offset \* num\_of\_constants) + wnaca];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h4\_i] = 1.00000+ (STATES[(offset \* num\_of\_states) + nai]/CONSTANTS[(offset \* num\_of\_constants) + kna1])\*(1.00000+STATES[(offset \* num\_of\_states) + nai]/CONSTANTS[(offset \* num\_of\_constants) + kna2]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + h5\_i] = ( STATES[(offset \* num\_of\_states) + nai]\*STATES[(offset \* num\_of\_states) + nai])/( ALGEBRAIC[(offset \* num\_of\_algebraic) + h4\_i]\*CONSTANTS[(offset \* num\_of\_constants) + kna1]\*CONSTANTS[(offset \* num\_of\_constants) + kna2]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h5\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + h2\_i]\*CONSTANTS[(offset \* num\_of\_constants) + wna];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h8\_i]\*CONSTANTS[(offset \* num\_of\_constants) + h11\_i]\*CONSTANTS[(offset \* num\_of\_constants) + wna];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h9\_i] = 1.00000/ALGEBRAIC[(offset \* num\_of\_algebraic) + h7\_i];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k3p\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h9\_i]\*CONSTANTS[(offset \* num\_of\_constants) + wca];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + k3p\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k3pp\_i];

ALGEBRAIC[(offset \* num\_of\_algebraic) + hca] = exp(( CONSTANTS[(offset \* num\_of\_constants) + qca]\*STATES[(offset \* num\_of\_states) + V]\*CONSTANTS[(offset \* num\_of\_constants) + F])/( CONSTANTS[(offset \* num\_of\_constants) + R]\*CONSTANTS[(offset \* num\_of\_constants) + T]));

ALGEBRAIC[(offset \* num\_of\_algebraic) + h3\_i] = 1.00000/ALGEBRAIC[(offset \* num\_of\_algebraic) + h1\_i];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k4p\_i] = ( ALGEBRAIC[(offset \* num\_of\_algebraic) + h3\_i]\*CONSTANTS[(offset \* num\_of\_constants) + wca])/ALGEBRAIC[(offset \* num\_of\_algebraic) + hca];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + k4p\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k4pp\_i];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h6\_i] = 1.00000/ALGEBRAIC[(offset \* num\_of\_algebraic) + h4\_i];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h6\_i]\*STATES[(offset \* num\_of\_states) + cai]\*CONSTANTS[(offset \* num\_of\_constants) + kcaon];

ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_i] = CONSTANTS[(offset \* num\_of\_constants) + k2\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_i]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_i])+ CONSTANTS[(offset \* num\_of\_constants) + k5\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_i]\*(CONSTANTS[(offset \* num\_of\_constants) + k2\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_i] = CONSTANTS[(offset \* num\_of\_constants) + k1\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_i]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_i]+CONSTANTS[(offset \* num\_of\_constants) + k5\_i])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_i]\*(CONSTANTS[(offset \* num\_of\_constants) + k1\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_i] = CONSTANTS[(offset \* num\_of\_constants) + k1\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_i]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_i])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_i]\*(CONSTANTS[(offset \* num\_of\_constants) + k2\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_i] = CONSTANTS[(offset \* num\_of\_constants) + k2\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_i]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_i]+CONSTANTS[(offset \* num\_of\_constants) + k5\_i])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_i]\*CONSTANTS[(offset \* num\_of\_constants) + k5\_i]\*(CONSTANTS[(offset \* num\_of\_constants) + k1\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E1\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_i]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E2\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_i]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E3\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_i]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E4\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_i]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + JncxNa\_i] = ( 3.00000\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + E4\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_i] - ALGEBRAIC[(offset \* num\_of\_algebraic) + E1\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_i])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + E3\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k4pp\_i]) - ALGEBRAIC[(offset \* num\_of\_algebraic) + E2\_i]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k3pp\_i];

ALGEBRAIC[(offset \* num\_of\_algebraic) + JncxCa\_i] = ALGEBRAIC[(offset \* num\_of\_algebraic) + E2\_i]\*CONSTANTS[(offset \* num\_of\_constants) + k2\_i] - ALGEBRAIC[(offset \* num\_of\_algebraic) + E1\_i]\*CONSTANTS[(offset \* num\_of\_constants) + k1\_i];

ALGEBRAIC[(offset \* num\_of\_algebraic) + INaCa\_i] = 0.800000\*CONSTANTS[(offset \* num\_of\_constants) + Gncx]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + allo\_i]\*( CONSTANTS[(offset \* num\_of\_constants) + zna]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + JncxNa\_i]+ CONSTANTS[(offset \* num\_of\_constants) + zca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + JncxCa\_i]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + INab] = ( CONSTANTS[(offset \* num\_of\_constants) + PNab]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vffrt]\*( STATES[(offset \* num\_of\_states) + nai]\*exp(ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - CONSTANTS[(offset \* num\_of\_constants) + nao]))/(exp(ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + JdiffNa] = (STATES[(offset \* num\_of\_states) + nass] - STATES[(offset \* num\_of\_states) + nai])/2.00000;

ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaNa] = ( 1.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vffrt]\*( 0.750000\*STATES[(offset \* num\_of\_states) + nass]\*exp( 1.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 0.750000\*CONSTANTS[(offset \* num\_of\_constants) + nao]))/(exp( 1.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaNa] = (1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + fICaLp])\*CONSTANTS[(offset \* num\_of\_constants) + PCaNa]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaNa]\*STATES[(offset \* num\_of\_states) + d]\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + f]\*(1.00000 - STATES[(offset \* num\_of\_states) + nca])+ STATES[(offset \* num\_of\_states) + jca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + fca]\*STATES[(offset \* num\_of\_states) + nca])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + fICaLp]\*CONSTANTS[(offset \* num\_of\_constants) + PCaNap]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + PhiCaNa]\*STATES[(offset \* num\_of\_states) + d]\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + fp]\*(1.00000 - STATES[(offset \* num\_of\_states) + nca])+ STATES[(offset \* num\_of\_states) + jca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + fcap]\*STATES[(offset \* num\_of\_states) + nca]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + allo\_ss] = 1.00000/(1.00000+pow(CONSTANTS[(offset \* num\_of\_constants) + KmCaAct]/STATES[(offset \* num\_of\_states) + cass], 2.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + h7\_ss] = 1.00000+ (CONSTANTS[(offset \* num\_of\_constants) + nao]/CONSTANTS[(offset \* num\_of\_constants) + kna3])\*(1.00000+1.00000/ALGEBRAIC[(offset \* num\_of\_algebraic) + hna]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + h8\_ss] = CONSTANTS[(offset \* num\_of\_constants) + nao]/( CONSTANTS[(offset \* num\_of\_constants) + kna3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + hna]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + h7\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + k3pp\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h8\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + wnaca];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h1\_ss] = 1.00000+ (STATES[(offset \* num\_of\_states) + nass]/CONSTANTS[(offset \* num\_of\_constants) + kna3])\*(1.00000+ALGEBRAIC[(offset \* num\_of\_algebraic) + hna]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + h2\_ss] = ( STATES[(offset \* num\_of\_states) + nass]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + hna])/( CONSTANTS[(offset \* num\_of\_constants) + kna3]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + h1\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + k4pp\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h2\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + wnaca];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h4\_ss] = 1.00000+ (STATES[(offset \* num\_of\_states) + nass]/CONSTANTS[(offset \* num\_of\_constants) + kna1])\*(1.00000+STATES[(offset \* num\_of\_states) + nass]/CONSTANTS[(offset \* num\_of\_constants) + kna2]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + h5\_ss] = ( STATES[(offset \* num\_of\_states) + nass]\*STATES[(offset \* num\_of\_states) + nass])/( ALGEBRAIC[(offset \* num\_of\_algebraic) + h4\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + kna1]\*CONSTANTS[(offset \* num\_of\_constants) + kna2]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h5\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + h2\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + wna];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h8\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + h11\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + wna];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h9\_ss] = 1.00000/ALGEBRAIC[(offset \* num\_of\_algebraic) + h7\_ss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k3p\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h9\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + wca];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + k3p\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k3pp\_ss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h3\_ss] = 1.00000/ALGEBRAIC[(offset \* num\_of\_algebraic) + h1\_ss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k4p\_ss] = ( ALGEBRAIC[(offset \* num\_of\_algebraic) + h3\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + wca])/ALGEBRAIC[(offset \* num\_of\_algebraic) + hca];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + k4p\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k4pp\_ss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + h6\_ss] = 1.00000/ALGEBRAIC[(offset \* num\_of\_algebraic) + h4\_ss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + h6\_ss]\*STATES[(offset \* num\_of\_states) + cass]\*CONSTANTS[(offset \* num\_of\_constants) + kcaon];

ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_ss] = CONSTANTS[(offset \* num\_of\_constants) + k2\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_ss]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_ss])+ CONSTANTS[(offset \* num\_of\_constants) + k5\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_ss]\*(CONSTANTS[(offset \* num\_of\_constants) + k2\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_ss] = CONSTANTS[(offset \* num\_of\_constants) + k1\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_ss]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_ss]+CONSTANTS[(offset \* num\_of\_constants) + k5\_ss])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_ss]\*(CONSTANTS[(offset \* num\_of\_constants) + k1\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_ss] = CONSTANTS[(offset \* num\_of\_constants) + k1\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_ss]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_ss])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k6\_ss]\*(CONSTANTS[(offset \* num\_of\_constants) + k2\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_ss] = CONSTANTS[(offset \* num\_of\_constants) + k2\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_ss]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + k4\_ss]+CONSTANTS[(offset \* num\_of\_constants) + k5\_ss])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + k3\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + k5\_ss]\*(CONSTANTS[(offset \* num\_of\_constants) + k1\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E1\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_ss]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E2\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_ss]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E3\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_ss]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + E4\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_ss]/(ALGEBRAIC[(offset \* num\_of\_algebraic) + x1\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x2\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x3\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + x4\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + JncxNa\_ss] = ( 3.00000\*( ALGEBRAIC[(offset \* num\_of\_algebraic) + E4\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k7\_ss] - ALGEBRAIC[(offset \* num\_of\_algebraic) + E1\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k8\_ss])+ ALGEBRAIC[(offset \* num\_of\_algebraic) + E3\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k4pp\_ss]) - ALGEBRAIC[(offset \* num\_of\_algebraic) + E2\_ss]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + k3pp\_ss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + JncxCa\_ss] = ALGEBRAIC[(offset \* num\_of\_algebraic) + E2\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + k2\_ss] - ALGEBRAIC[(offset \* num\_of\_algebraic) + E1\_ss]\*CONSTANTS[(offset \* num\_of\_constants) + k1\_ss];

ALGEBRAIC[(offset \* num\_of\_algebraic) + INaCa\_ss] = 0.200000\*CONSTANTS[(offset \* num\_of\_constants) + Gncx]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + allo\_ss]\*( CONSTANTS[(offset \* num\_of\_constants) + zna]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + JncxNa\_ss]+ CONSTANTS[(offset \* num\_of\_constants) + zca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + JncxCa\_ss]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + IpCa] = ( CONSTANTS[(offset \* num\_of\_constants) + GpCa]\*STATES[(offset \* num\_of\_states) + cai])/(CONSTANTS[(offset \* num\_of\_constants) + KmCap]+STATES[(offset \* num\_of\_states) + cai]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + ICab] = ( CONSTANTS[(offset \* num\_of\_constants) + PCab]\*4.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vffrt]\*( STATES[(offset \* num\_of\_states) + cai]\*exp( 2.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 0.341000\*CONSTANTS[(offset \* num\_of\_constants) + cao]))/(exp( 2.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + vfrt]) - 1.00000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jdiff] = (STATES[(offset \* num\_of\_states) + cass] - STATES[(offset \* num\_of\_states) + cai])/0.200000;

ALGEBRAIC[(offset \* num\_of\_algebraic) + fJrelp] = 1.00000/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + KmCaMK]/ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKa]);

//cvar starts here

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel] = CONSTANTS[(offset \* num\_of\_constants) + Jrel\_scale] \* ( (1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + fJrelp])\*STATES[(offset \* num\_of\_states) + Jrelnp]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + fJrelp]\*STATES[(offset \* num\_of\_states) + Jrelp]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Bcass] = 1.00000/(1.00000+( CONSTANTS[(offset \* num\_of\_constants) + BSRmax]\*CONSTANTS[(offset \* num\_of\_constants) + KmBSR])/pow(CONSTANTS[(offset \* num\_of\_constants) + KmBSR]+STATES[(offset \* num\_of\_states) + cass], 2.00000)+( CONSTANTS[(offset \* num\_of\_constants) + BSLmax]\*CONSTANTS[(offset \* num\_of\_constants) + KmBSL])/pow(CONSTANTS[(offset \* num\_of\_constants) + KmBSL]+STATES[(offset \* num\_of\_states) + cass], 2.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jupnp] = ( CONSTANTS[(offset \* num\_of\_constants) + upScale]\*0.00437500\*STATES[(offset \* num\_of\_states) + cai])/(STATES[(offset \* num\_of\_states) + cai]+0.000920000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jupp] = ( CONSTANTS[(offset \* num\_of\_constants) + upScale]\*2.75000\*0.00437500\*STATES[(offset \* num\_of\_states) + cai])/((STATES[(offset \* num\_of\_states) + cai]+0.000920000) - 0.000170000);

ALGEBRAIC[(offset \* num\_of\_algebraic) + fJupp] = 1.00000/(1.00000+CONSTANTS[(offset \* num\_of\_constants) + KmCaMK]/ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKa]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jleak] = CONSTANTS[(offset \* num\_of\_constants) + Jleak\_scale] \* ( 0.00393750\*STATES[(offset \* num\_of\_states) + cansr])/15.0000;

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jup] = CONSTANTS[(offset \* num\_of\_constants) + Jup\_scale] \* ( ( (1.00000 - ALGEBRAIC[(offset \* num\_of\_algebraic) + fJupp])\*ALGEBRAIC[(offset \* num\_of\_algebraic) + Jupnp]+ ALGEBRAIC[(offset \* num\_of\_algebraic) + fJupp]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + Jupp]) - ALGEBRAIC[(offset \* num\_of\_algebraic) + Jleak]);

ALGEBRAIC[(offset \* num\_of\_algebraic) + Bcai] = 1.00000/(1.00000+( CONSTANTS[(offset \* num\_of\_constants) + cmdnmax]\*CONSTANTS[(offset \* num\_of\_constants) + kmcmdn])/pow(CONSTANTS[(offset \* num\_of\_constants) + kmcmdn]+STATES[(offset \* num\_of\_states) + cai], 2.00000)+( CONSTANTS[(offset \* num\_of\_constants) + trpnmax]\*CONSTANTS[(offset \* num\_of\_constants) + kmtrpn])/pow(CONSTANTS[(offset \* num\_of\_constants) + kmtrpn]+STATES[(offset \* num\_of\_states) + cai], 2.00000));

ALGEBRAIC[(offset \* num\_of\_algebraic) + Jtr] = CONSTANTS[(offset \* num\_of\_constants) + Jtr\_scale] \* (STATES[(offset \* num\_of\_states) + cansr] - STATES[(offset \* num\_of\_states) + cajsr])/100.000;

//cvar ends here

ALGEBRAIC[(offset \* num\_of\_algebraic) + Bcajsr] = 1.00000/(1.00000+( CONSTANTS[(offset \* num\_of\_constants) + csqnmax]\*CONSTANTS[(offset \* num\_of\_constants) + kmcsqn])/pow(CONSTANTS[(offset \* num\_of\_constants) + kmcsqn]+STATES[(offset \* num\_of\_states) + cajsr], 2.00000));

RATES[(offset \* num\_of\_rates) + hL] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + hLss] - STATES[(offset \* num\_of\_states) + hL])/CONSTANTS[(offset \* num\_of\_constants) + thL];

RATES[(offset \* num\_of\_rates) + hLp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + hLssp] - STATES[(offset \* num\_of\_states) + hLp])/CONSTANTS[(offset \* num\_of\_constants) + thLp];

RATES[(offset \* num\_of\_rates) + m] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + mss] - STATES[(offset \* num\_of\_states) + m])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tm];

RATES[(offset \* num\_of\_rates) + hf] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + hss] - STATES[(offset \* num\_of\_states) + hf])/ALGEBRAIC[(offset \* num\_of\_algebraic) + thf];

RATES[(offset \* num\_of\_rates) + hs] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + hss] - STATES[(offset \* num\_of\_states) + hs])/ALGEBRAIC[(offset \* num\_of\_algebraic) + ths];

RATES[(offset \* num\_of\_rates) + a] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + ass] - STATES[(offset \* num\_of\_states) + a])/ALGEBRAIC[(offset \* num\_of\_algebraic) + ta];

RATES[(offset \* num\_of\_rates) + d] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + dss] - STATES[(offset \* num\_of\_states) + d])/ALGEBRAIC[(offset \* num\_of\_algebraic) + td];

RATES[(offset \* num\_of\_rates) + ff] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - STATES[(offset \* num\_of\_states) + ff])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tff];

RATES[(offset \* num\_of\_rates) + fs] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - STATES[(offset \* num\_of\_states) + fs])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tfs];

RATES[(offset \* num\_of\_rates) + jca] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - STATES[(offset \* num\_of\_states) + jca])/CONSTANTS[(offset \* num\_of\_constants) + tjca];

RATES[(offset \* num\_of\_rates) + nca] = ALGEBRAIC[(offset \* num\_of\_algebraic) + anca]\*CONSTANTS[(offset \* num\_of\_constants) + k2n] - STATES[(offset \* num\_of\_states) + nca]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + km2n];

RATES[(offset \* num\_of\_rates) + xrf] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + xrss] - STATES[(offset \* num\_of\_states) + xrf])/ALGEBRAIC[(offset \* num\_of\_algebraic) + txrf];

RATES[(offset \* num\_of\_rates) + xrs] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + xrss] - STATES[(offset \* num\_of\_states) + xrs])/ALGEBRAIC[(offset \* num\_of\_algebraic) + txrs];

RATES[(offset \* num\_of\_rates) + xs1] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + xs1ss] - STATES[(offset \* num\_of\_states) + xs1])/ALGEBRAIC[(offset \* num\_of\_algebraic) + txs1];

RATES[(offset \* num\_of\_rates) + xk1] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + xk1ss] - STATES[(offset \* num\_of\_states) + xk1])/ALGEBRAIC[(offset \* num\_of\_algebraic) + txk1];

RATES[(offset \* num\_of\_rates) + j] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + jss] - STATES[(offset \* num\_of\_states) + j])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tj];

RATES[(offset \* num\_of\_rates) + ap] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + assp] - STATES[(offset \* num\_of\_states) + ap])/ALGEBRAIC[(offset \* num\_of\_algebraic) + ta];

RATES[(offset \* num\_of\_rates) + fcaf] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - STATES[(offset \* num\_of\_states) + fcaf])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcaf];

RATES[(offset \* num\_of\_rates) + fcas] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - STATES[(offset \* num\_of\_states) + fcas])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcas];

RATES[(offset \* num\_of\_rates) + ffp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - STATES[(offset \* num\_of\_states) + ffp])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tffp];

RATES[(offset \* num\_of\_rates) + xs2] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + xs2ss] - STATES[(offset \* num\_of\_states) + xs2])/ALGEBRAIC[(offset \* num\_of\_algebraic) + txs2];

RATES[(offset \* num\_of\_rates) + CaMKt] = CONSTANTS[(offset \* num\_of\_constants) + aCaMK]\*ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKb]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + CaMKb]+STATES[(offset \* num\_of\_states) + CaMKt]) - CONSTANTS[(offset \* num\_of\_constants) + bCaMK]\*STATES[(offset \* num\_of\_states) + CaMKt];

RATES[(offset \* num\_of\_rates) + hsp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + hssp] - STATES[(offset \* num\_of\_states) + hsp])/ALGEBRAIC[(offset \* num\_of\_algebraic) + thsp];

RATES[(offset \* num\_of\_rates) + jp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + jss] - STATES[(offset \* num\_of\_states) + jp])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tjp];

RATES[(offset \* num\_of\_rates) + mL] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + mLss] - STATES[(offset \* num\_of\_states) + mL])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tmL];

RATES[(offset \* num\_of\_rates) + fcafp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - STATES[(offset \* num\_of\_states) + fcafp])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcafp];

RATES[(offset \* num\_of\_rates) + iF] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - STATES[(offset \* num\_of\_states) + iF])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tiF];

RATES[(offset \* num\_of\_rates) + iS] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - STATES[(offset \* num\_of\_states) + iS])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tiS];

RATES[(offset \* num\_of\_rates) + iFp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - STATES[(offset \* num\_of\_states) + iFp])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tiFp];

RATES[(offset \* num\_of\_rates) + iSp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - STATES[(offset \* num\_of\_states) + iSp])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tiSp];

RATES[(offset \* num\_of\_rates) + Jrelnp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_inf] - STATES[(offset \* num\_of\_states) + Jrelnp])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_rel];

RATES[(offset \* num\_of\_rates) + Jrelp] = (ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_infp] - STATES[(offset \* num\_of\_states) + Jrelp])/ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_relp];

RATES[(offset \* num\_of\_rates) + ki] = ( - ((ALGEBRAIC[(offset \* num\_of\_algebraic) + Ito]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IKr]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IKs]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IK1]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IKb]+ALGEBRAIC[(offset \* num\_of\_algebraic) + Istim]) - 2.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + INaK])\*CONSTANTS[(offset \* num\_of\_constants) + cm]\*CONSTANTS[(offset \* num\_of\_constants) + Acap])/( CONSTANTS[(offset \* num\_of\_constants) + F]\*CONSTANTS[(offset \* num\_of\_constants) + vmyo])+( ALGEBRAIC[(offset \* num\_of\_algebraic) + JdiffK]\*CONSTANTS[(offset \* num\_of\_constants) + vss])/CONSTANTS[(offset \* num\_of\_constants) + vmyo];

RATES[(offset \* num\_of\_rates) + kss] = ( - ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaK]\*CONSTANTS[(offset \* num\_of\_constants) + cm]\*CONSTANTS[(offset \* num\_of\_constants) + Acap])/( CONSTANTS[(offset \* num\_of\_constants) + F]\*CONSTANTS[(offset \* num\_of\_constants) + vss]) - ALGEBRAIC[(offset \* num\_of\_algebraic) + JdiffK];

RATES[(offset \* num\_of\_rates) + nai] = ( - (ALGEBRAIC[(offset \* num\_of\_algebraic) + INa]+ALGEBRAIC[(offset \* num\_of\_algebraic) + INaL]+ 3.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + INaCa\_i]+ 3.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + INaK]+ALGEBRAIC[(offset \* num\_of\_algebraic) + INab])\*CONSTANTS[(offset \* num\_of\_constants) + Acap]\*CONSTANTS[(offset \* num\_of\_constants) + cm])/( CONSTANTS[(offset \* num\_of\_constants) + F]\*CONSTANTS[(offset \* num\_of\_constants) + vmyo])+( ALGEBRAIC[(offset \* num\_of\_algebraic) + JdiffNa]\*CONSTANTS[(offset \* num\_of\_constants) + vss])/CONSTANTS[(offset \* num\_of\_constants) + vmyo];

RATES[(offset \* num\_of\_rates) + nass] = ( - (ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaNa]+ 3.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + INaCa\_ss])\*CONSTANTS[(offset \* num\_of\_constants) + cm]\*CONSTANTS[(offset \* num\_of\_constants) + Acap])/( CONSTANTS[(offset \* num\_of\_constants) + F]\*CONSTANTS[(offset \* num\_of\_constants) + vss]) - ALGEBRAIC[(offset \* num\_of\_algebraic) + JdiffNa];

RATES[(offset \* num\_of\_rates) + V] = - (ALGEBRAIC[(offset \* num\_of\_algebraic) + INa]+ALGEBRAIC[(offset \* num\_of\_algebraic) + INaL]+ALGEBRAIC[(offset \* num\_of\_algebraic) + Ito]+ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaL]+ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaNa]+ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaK]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IKr]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IKs]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IK1]+ALGEBRAIC[(offset \* num\_of\_algebraic) + INaCa\_i]+ALGEBRAIC[(offset \* num\_of\_algebraic) + INaCa\_ss]+ALGEBRAIC[(offset \* num\_of\_algebraic) + INaK]+ALGEBRAIC[(offset \* num\_of\_algebraic) + INab]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IKb]+ALGEBRAIC[(offset \* num\_of\_algebraic) + IpCa]+ALGEBRAIC[(offset \* num\_of\_algebraic) + ICab]+ALGEBRAIC[(offset \* num\_of\_algebraic) + Istim]);

RATES[(offset \* num\_of\_rates) + cass] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Bcass]\*((( - (ALGEBRAIC[(offset \* num\_of\_algebraic) + ICaL] - 2.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + INaCa\_ss])\*CONSTANTS[(offset \* num\_of\_constants) + cm]\*CONSTANTS[(offset \* num\_of\_constants) + Acap])/( 2.00000\*CONSTANTS[(offset \* num\_of\_constants) + F]\*CONSTANTS[(offset \* num\_of\_constants) + vss])+( ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel]\*CONSTANTS[(offset \* num\_of\_constants) + vjsr])/CONSTANTS[(offset \* num\_of\_constants) + vss]) - ALGEBRAIC[(offset \* num\_of\_algebraic) + Jdiff]);

RATES[(offset \* num\_of\_rates) + cai] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Bcai]\*((( - ((ALGEBRAIC[(offset \* num\_of\_algebraic) + IpCa]+ALGEBRAIC[(offset \* num\_of\_algebraic) + ICab]) - 2.00000\*ALGEBRAIC[(offset \* num\_of\_algebraic) + INaCa\_i])\*CONSTANTS[(offset \* num\_of\_constants) + cm]\*CONSTANTS[(offset \* num\_of\_constants) + Acap])/( 2.00000\*CONSTANTS[(offset \* num\_of\_constants) + F]\*CONSTANTS[(offset \* num\_of\_constants) + vmyo]) - ( ALGEBRAIC[(offset \* num\_of\_algebraic) + Jup]\*CONSTANTS[(offset \* num\_of\_constants) + vnsr])/CONSTANTS[(offset \* num\_of\_constants) + vmyo])+( ALGEBRAIC[(offset \* num\_of\_algebraic) + Jdiff]\*CONSTANTS[(offset \* num\_of\_constants) + vss])/CONSTANTS[(offset \* num\_of\_constants) + vmyo]);

RATES[(offset \* num\_of\_rates) + cansr] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Jup] - ( ALGEBRAIC[(offset \* num\_of\_algebraic) + Jtr]\*CONSTANTS[(offset \* num\_of\_constants) + vjsr])/CONSTANTS[(offset \* num\_of\_constants) + vnsr];

RATES[(offset \* num\_of\_rates) + cajsr] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Bcajsr]\*(ALGEBRAIC[(offset \* num\_of\_algebraic) + Jtr] - ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel]);

}

\_\_device\_\_ void solveEuler( double \*STATES, double \*RATES, double dt, int offset){

int num\_of\_states = 41;

int num\_of\_rates = 41;

STATES[(offset \* num\_of\_states) +V] = STATES[(offset \* num\_of\_states) + V] + RATES[(offset \* num\_of\_rates) + V] \* dt;

STATES[(offset \* num\_of\_states) + CaMKt] = STATES[(offset \* num\_of\_states) + CaMKt] + RATES[(offset \* num\_of\_rates) + CaMKt] \* dt;

STATES[(offset \* num\_of\_states) + cass] = STATES[(offset \* num\_of\_states) + cass] + RATES[(offset \* num\_of\_rates) + cass] \* dt;

STATES[(offset \* num\_of\_states) + nai] = STATES[(offset \* num\_of\_states) + nai] + RATES[(offset \* num\_of\_rates) + nai] \* dt;

STATES[(offset \* num\_of\_states) + nass] = STATES[(offset \* num\_of\_states) + nass] + RATES[(offset \* num\_of\_rates) + nass] \* dt;

STATES[(offset \* num\_of\_states) + ki] = STATES[(offset \* num\_of\_states) + ki] + RATES[(offset \* num\_of\_rates) + ki] \* dt;

STATES[(offset \* num\_of\_states) + kss] = STATES[(offset \* num\_of\_states) + kss] + RATES[(offset \* num\_of\_rates) + kss] \* dt;

STATES[(offset \* num\_of\_states) + cansr] = STATES[(offset \* num\_of\_states) + cansr] + RATES[(offset \* num\_of\_rates) + cansr] \* dt;

STATES[(offset \* num\_of\_states) + cajsr] = STATES[(offset \* num\_of\_states) + cajsr] + RATES[(offset \* num\_of\_rates) + cajsr] \* dt;

STATES[(offset \* num\_of\_states) + cai] = STATES[(offset \* num\_of\_states) + cai] + RATES[(offset \* num\_of\_rates) + cai] \* dt;

STATES[(offset \* num\_of\_states) + m] = STATES[(offset \* num\_of\_states) + m] + RATES[(offset \* num\_of\_rates) + m] \* dt;

STATES[(offset \* num\_of\_states) + hf] = STATES[(offset \* num\_of\_states) + hf] + RATES[(offset \* num\_of\_rates) + hf] \* dt;

STATES[(offset \* num\_of\_states) + hs] = STATES[(offset \* num\_of\_states) + hs] + RATES[(offset \* num\_of\_rates) + hs] \* dt;

STATES[(offset \* num\_of\_states) + j] = STATES[(offset \* num\_of\_states) + j] + RATES[(offset \* num\_of\_rates) + j] \* dt;

STATES[(offset \* num\_of\_states) + hsp] = STATES[(offset \* num\_of\_states) + hsp] + RATES[(offset \* num\_of\_rates) + hsp] \* dt;

STATES[(offset \* num\_of\_states) + jp] = STATES[(offset \* num\_of\_states) + jp] + RATES[(offset \* num\_of\_rates) + jp] \* dt;

STATES[(offset \* num\_of\_states) + mL] = STATES[(offset \* num\_of\_states) + mL] + RATES[(offset \* num\_of\_rates) + mL] \* dt;

STATES[(offset \* num\_of\_states) + hL] = STATES[(offset \* num\_of\_states) + hL] + RATES[(offset \* num\_of\_rates) + hL] \* dt;

STATES[(offset \* num\_of\_states) + hLp] = STATES[(offset \* num\_of\_states) + hLp] + RATES[(offset \* num\_of\_rates) + hLp] \* dt;

STATES[(offset \* num\_of\_states) + a] = STATES[(offset \* num\_of\_states) + a] + RATES[(offset \* num\_of\_rates) + a] \* dt;

STATES[(offset \* num\_of\_states) + iF] = STATES[(offset \* num\_of\_states) + iF] + RATES[(offset \* num\_of\_rates) + iF] \* dt;

STATES[(offset \* num\_of\_states) + iS] = STATES[(offset \* num\_of\_states) + iS] + RATES[(offset \* num\_of\_rates) + iS] \* dt;

STATES[(offset \* num\_of\_states) + ap] = STATES[(offset \* num\_of\_states) + ap] + RATES[(offset \* num\_of\_rates) + ap] \* dt;

STATES[(offset \* num\_of\_states) + iFp] = STATES[(offset \* num\_of\_states) + iFp] + RATES[(offset \* num\_of\_rates) + iFp] \* dt;

STATES[(offset \* num\_of\_states) + iSp] = STATES[(offset \* num\_of\_states) + iSp] + RATES[(offset \* num\_of\_rates) + iSp] \* dt;

STATES[(offset \* num\_of\_states) + d] = STATES[(offset \* num\_of\_states) + d] + RATES[(offset \* num\_of\_rates) + d] \* dt;

STATES[(offset \* num\_of\_states) + ff] = STATES[(offset \* num\_of\_states) + ff] + RATES[(offset \* num\_of\_rates) + ff] \* dt;

STATES[(offset \* num\_of\_states) + fs] = STATES[(offset \* num\_of\_states) + fs] + RATES[(offset \* num\_of\_rates) + fs] \* dt;

STATES[(offset \* num\_of\_states) + fcaf] = STATES[(offset \* num\_of\_states) + fcaf] + RATES[(offset \* num\_of\_rates) + fcaf] \* dt;

STATES[(offset \* num\_of\_states) + fcas] = STATES[(offset \* num\_of\_states) + fcas] + RATES[(offset \* num\_of\_rates) + fcas] \* dt;

STATES[(offset \* num\_of\_states) + jca] = STATES[(offset \* num\_of\_states) + jca] + RATES[(offset \* num\_of\_rates) + jca] \* dt;

STATES[(offset \* num\_of\_states) + ffp] = STATES[(offset \* num\_of\_states) + ffp] + RATES[(offset \* num\_of\_rates) + ffp] \* dt;

STATES[(offset \* num\_of\_states) + fcafp] = STATES[(offset \* num\_of\_states) + fcafp] + RATES[(offset \* num\_of\_rates) + fcafp] \* dt;

STATES[(offset \* num\_of\_states) + nca] = STATES[(offset \* num\_of\_states) + nca] + RATES[(offset \* num\_of\_rates) + nca] \* dt;

STATES[(offset \* num\_of\_states) + xrf] = STATES[(offset \* num\_of\_states) + xrf] + RATES[(offset \* num\_of\_rates) + xrf] \* dt;

STATES[(offset \* num\_of\_states) + xrs] = STATES[(offset \* num\_of\_states) + xrs] + RATES[(offset \* num\_of\_rates) + xrs] \* dt;

STATES[(offset \* num\_of\_states) + xs1] = STATES[(offset \* num\_of\_states) + xs1] + RATES[(offset \* num\_of\_rates) + xs1] \* dt;

STATES[(offset \* num\_of\_states) + xs2] = STATES[(offset \* num\_of\_states) + xs2] + RATES[(offset \* num\_of\_rates) + xs2] \* dt;

STATES[(offset \* num\_of\_states) + xk1] = STATES[(offset \* num\_of\_states) + xk1] + RATES[(offset \* num\_of\_rates) + xk1] \* dt;

STATES[(offset \* num\_of\_states) + Jrelnp] = STATES[(offset \* num\_of\_states) + Jrelnp] + RATES[(offset \* num\_of\_rates) + Jrelnp] \* dt;

STATES[(offset \* num\_of\_states) + Jrelp] = STATES[(offset \* num\_of\_states) + Jrelp] + RATES[(offset \* num\_of\_rates) + Jrelp] \* dt;

}

\_\_device\_\_ void solveAnalytical(double \*CONSTANTS, double \*STATES, double \*ALGEBRAIC, double \*RATES, double dt, int offset)

{

int num\_of\_constants = 145;

int num\_of\_states = 41;

int num\_of\_algebraic = 199;

int num\_of\_rates = 41;

// #ifdef EULER // moved as its own function

// #else

////==============

////Exact solution

////==============

////INa

STATES[(offset \* num\_of\_states) + m] = ALGEBRAIC[(offset \* num\_of\_algebraic) + mss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + mss] - STATES[(offset \* num\_of\_states) + m]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tm]);

STATES[(offset \* num\_of\_states) + hf] = ALGEBRAIC[(offset \* num\_of\_algebraic) + hss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + hss] - STATES[(offset \* num\_of\_states) + hf]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + thf]);

STATES[(offset \* num\_of\_states) + hs] = ALGEBRAIC[(offset \* num\_of\_algebraic) + hss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + hss] - STATES[(offset \* num\_of\_states) + hs]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + ths]);

STATES[(offset \* num\_of\_states) + j] = ALGEBRAIC[(offset \* num\_of\_algebraic) + jss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + jss] - STATES[(offset \* num\_of\_states) + j]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tj]);

STATES[(offset \* num\_of\_states) + hsp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + hssp] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + hssp] - STATES[(offset \* num\_of\_states) + hsp]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + thsp]);

STATES[(offset \* num\_of\_states) + jp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + jss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + jss] - STATES[(offset \* num\_of\_states) + jp]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tjp]);

STATES[(offset \* num\_of\_states) + mL] = ALGEBRAIC[(offset \* num\_of\_algebraic) + mLss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + mLss] - STATES[(offset \* num\_of\_states) + mL]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tmL]);

STATES[(offset \* num\_of\_states) + hL] = ALGEBRAIC[(offset \* num\_of\_algebraic) + hLss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + hLss] - STATES[(offset \* num\_of\_states) + hL]) \* exp(-dt / CONSTANTS[(offset \* num\_of\_constants) + thL]);

STATES[(offset \* num\_of\_states) + hLp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + hLssp] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + hLssp] - STATES[(offset \* num\_of\_states) + hLp]) \* exp(-dt / CONSTANTS[(offset \* num\_of\_constants) + thLp]);

////Ito

STATES[(offset \* num\_of\_states) + a] = ALGEBRAIC[(offset \* num\_of\_algebraic) + ass] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + ass] - STATES[(offset \* num\_of\_states) + a]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + ta]);

STATES[(offset \* num\_of\_states) + iF] = ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - STATES[(offset \* num\_of\_states) + iF]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tiF]);

STATES[(offset \* num\_of\_states) + iS] = ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - STATES[(offset \* num\_of\_states) + iS]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tiS]);

STATES[(offset \* num\_of\_states) + ap] = ALGEBRAIC[(offset \* num\_of\_algebraic) + assp] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + assp] - STATES[(offset \* num\_of\_states) + ap]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + ta]);

STATES[(offset \* num\_of\_states) + iFp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - STATES[(offset \* num\_of\_states) + iFp]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tiFp]);

STATES[(offset \* num\_of\_states) + iSp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + iss] - STATES[(offset \* num\_of\_states) + iSp]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tiSp]);

////ICaL

STATES[(offset \* num\_of\_states) + d] = ALGEBRAIC[(offset \* num\_of\_algebraic) + dss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + dss] - STATES[(offset \* num\_of\_states) + d]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + td]);

STATES[(offset \* num\_of\_states) + ff] = ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - STATES[(offset \* num\_of\_states) + ff]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tff]);

STATES[(offset \* num\_of\_states) + fs] = ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - STATES[(offset \* num\_of\_states) + fs]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tfs]);

STATES[(offset \* num\_of\_states) + fcaf] = ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - STATES[(offset \* num\_of\_states) + fcaf]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcaf]);

STATES[(offset \* num\_of\_states) + fcas] = ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - STATES[(offset \* num\_of\_states) + fcas]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcas]);

STATES[(offset \* num\_of\_states) + jca] = ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - STATES[(offset \* num\_of\_states) + jca]) \* exp(- dt / CONSTANTS[(offset \* num\_of\_constants) + tjca]);

STATES[(offset \* num\_of\_states) + ffp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + fss] - STATES[(offset \* num\_of\_states) + ffp]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tffp]);

STATES[(offset \* num\_of\_states) + fcafp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + fcass] - STATES[(offset \* num\_of\_states) + fcafp]) \* exp(-d / ALGEBRAIC[(offset \* num\_of\_algebraic) + tfcafp]);

STATES[(offset \* num\_of\_states) + nca] = ALGEBRAIC[(offset \* num\_of\_algebraic) + anca] \* CONSTANTS[(offset \* num\_of\_constants) + k2n] / ALGEBRAIC[(offset \* num\_of\_algebraic) + km2n] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + anca] \* CONSTANTS[(offset \* num\_of\_constants) + k2n] / ALGEBRAIC[(offset \* num\_of\_algebraic) + km2n] - STATES[(offset \* num\_of\_states) + nca]) \* exp(-ALGEBRAIC[(offset \* num\_of\_algebraic) + km2n] \* dt);

////IKr

STATES[(offset \* num\_of\_states) + xrf] = ALGEBRAIC[(offset \* num\_of\_algebraic) + xrss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + xrss] - STATES[(offset \* num\_of\_states) + xrf]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + txrf]);

STATES[(offset \* num\_of\_states) + xrs] = ALGEBRAIC[(offset \* num\_of\_algebraic) + xrss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + xrss] - STATES[(offset \* num\_of\_states) + xrs]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + txrs]);

////IKs

STATES[(offset \* num\_of\_states) + xs1] = ALGEBRAIC[(offset \* num\_of\_algebraic) + xs1ss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + xs1ss] - STATES[(offset \* num\_of\_states) + xs1]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + txs1]);

STATES[(offset \* num\_of\_states) + xs2] = ALGEBRAIC[(offset \* num\_of\_algebraic) + xs2ss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + xs2ss] - STATES[(offset \* num\_of\_states) + xs2]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + txs2]);

////IK1

STATES[(offset \* num\_of\_states) + xk1] = ALGEBRAIC[(offset \* num\_of\_algebraic) + xk1ss] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + xk1ss] - STATES[(offset \* num\_of\_states) + xk1]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + txk1]);

////INaCa

////INaK

////IKb

////INab

////ICab

///IpCa

////Diffusion fluxes

////RyR receptors

STATES[(offset \* num\_of\_states) + Jrelnp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_inf] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_inf] - STATES[(offset \* num\_of\_states) + Jrelnp]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_rel]);

STATES[(offset \* num\_of\_states) + Jrelp] = ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_infp] - (ALGEBRAIC[(offset \* num\_of\_algebraic) + Jrel\_infp] - STATES[(offset \* num\_of\_states) + Jrelp]) \* exp(-dt / ALGEBRAIC[(offset \* num\_of\_algebraic) + tau\_relp]);

////SERCA Pump

////Calcium translocation

//

////=============================

////Approximated solution (Euler)

////=============================

////ICaL

//STATES[jca] = STATES[jca] + RATES[jca] \* dt;

////CaMK

STATES[(offset \* num\_of\_states) + CaMKt] = STATES[(offset \* num\_of\_states) + CaMKt] + RATES[(offset \* num\_of\_rates) + CaMKt] \* dt;

////Membrane potential

STATES[(offset \* num\_of\_states) + V] = STATES[(offset \* num\_of\_states) + V] + RATES[(offset \* num\_of\_rates) + V] \* dt;

////Ion Concentrations and Buffers

STATES[(offset \* num\_of\_states) + nai] = STATES[(offset \* num\_of\_states) + nai] + RATES[(offset \* num\_of\_rates) + nai] \* dt;

STATES[(offset \* num\_of\_states) + nass] = STATES[(offset \* num\_of\_states) + nass] + RATES[(offset \* num\_of\_rates) + nass] \* dt;

STATES[(offset \* num\_of\_states) + ki] = STATES[(offset \* num\_of\_states) + ki] + RATES[(offset \* num\_of\_rates) + ki] \* dt;

STATES[(offset \* num\_of\_states) + kss] = STATES[(offset \* num\_of\_states) + kss] + RATES[(offset \* num\_of\_rates) + kss] \* dt;

STATES[(offset \* num\_of\_states) + cai] = STATES[(offset \* num\_of\_states) + cai] + RATES[(offset \* num\_of\_rates) + cai] \* dt;

STATES[(offset \* num\_of\_states) + cass] = STATES[(offset \* num\_of\_states) + cass] + RATES[(offset \* num\_of\_rates) + cass] \* dt;

STATES[(offset \* num\_of\_states) + cansr] = STATES[(offset \* num\_of\_states) + cansr] + RATES[(offset \* num\_of\_rates) + cansr] \* dt;

STATES[(offset \* num\_of\_states) + cajsr] = STATES[(offset \* num\_of\_states) + cajsr] + RATES[(offset \* num\_of\_rates) + cajsr] \* dt;

// #endif

}

\_\_device\_\_ double set\_time\_step(double TIME,

double time\_point,

double max\_time\_step,

double \*CONSTANTS,

double \*RATES,

double \*STATES,

double \*ALGEBRAIC,

int offset) {

double time\_step = 0.005;

int num\_of\_constants = 145;

int num\_of\_rates = 41;

if (TIME <= time\_point || (TIME - floor(TIME / CONSTANTS[BCL + (offset \* num\_of\_constants)]) \* CONSTANTS[BCL + (offset \* num\_of\_constants)]) <= time\_point) {

//printf("TIME <= time\_point ms\n");

return time\_step;

//printf("dV = %lf, time\_step = %lf\n",RATES[V] \* time\_step, time\_step);

}

else {

//printf("TIME > time\_point ms\n");

if (std::abs(RATES[V + (offset \* num\_of\_rates)] \* time\_step) <= 0.2) {//Slow changes in V

// printf("dV/dt <= 0.2\n");

time\_step = std::abs(0.8 / RATES[V + (offset \* num\_of\_rates)]);

//Make sure time\_step is between 0.005 and max\_time\_step

if (time\_step < 0.005) {

time\_step = 0.005;

}

else if (time\_step > max\_time\_step) {

time\_step = max\_time\_step;

}

//printf("dV = %lf, time\_step = %lf\n",std::abs(RATES[V] \* time\_step), time\_step);

}

else if (std::abs(RATES[V + (offset \* num\_of\_rates)] \* time\_step) >= 0.8) {//Fast changes in V

// printf("dV/dt >= 0.8\n");

time\_step = std::abs(0.2 / RATES[V + (offset \* num\_of\_rates)]);

while (std::abs(RATES[V + (offset \* num\_of\_rates)] \* time\_step) >= 0.8 &&

0.005 < time\_step &&

time\_step < max\_time\_step) {

time\_step = time\_step / 10.0;

// printf("dV = %lf, time\_step = %lf\n",std::abs(RATES[V] \* time\_step), time\_step);

}

}

// \_\_syncthreads();

return time\_step;

}

}

## Cellmodel.hpp

This file contains the general interface for the cell models. It declares common functions that all cellmodel should have. Cellmodel.hpp implemented as below:

#ifndef CELL\_HPP

#define CELL\_HPP

class Cellmodel

{

protected:

Cellmodel(){}

public:

unsigned short algebraic\_size;

unsigned short constants\_size;

unsigned short states\_size;

unsigned short gates\_size;

unsigned short current\_size;

unsigned short concs\_size;

double ALGEBRAIC[255];

double CONSTANTS[255];

double RATES[255];

double STATES[255];

char gates\_header[255];

unsigned short gates\_indices[255];

char current\_header[255];

unsigned short current\_indices[255];

char concs\_header[255];

unsigned short concs\_indices[255];

virtual ~Cellmodel() {}

virtual void initConsts() = 0;

virtual void initConsts(double type){}

virtual void initConsts(double type, double conc, double \*hill){}

virtual void initConsts(double type, double conc, double \*hill, bool is\_dutta){}

virtual void computeRates(double TIME, double \*CONSTANTS, double \*RATES, double \*STATES, double \*ALGEBRAIC) = 0;

virtual void solveAnalytical(double dt) {};

};

#endif

## enums/enum\_Ohara\_rudy\_2011.hpp

This header act as a ‘translation dictionary’ for each variable in the cell model file. Enumeration required to easily track each variable, so instead of looking at numbers, we looked at pre-defined variable names corresponds to each correct values in the cell model. The enumeration follows CellML’s description of each value and their function, as in below:

#ifndef EN\_OHARA\_RUDY\_2011\_HPP

#define EN\_OHARA\_RUDY\_2011\_HPP

enum E\_ALGEBRAIC\_T{

vffrt = 29,

vfrt = 39,

INa = 62,

INaL = 64,

Ito = 70,

ICaL = 81,

ICaNa = 82,

ICaK = 85,

IKr = 96,

IKs = 98,

IK1 = 100,

INaCa\_i = 132,

INaCa\_ss = 162,

INaK = 181,

INab = 184,

IKb = 183,

IpCa = 188,

ICab = 186,

Istim = 12,

CaMKb = 45,

CaMKa = 47,

JdiffNa = 187,

Jdiff = 189,

Jup = 196,

JdiffK = 185,

Jrel = 191,

Jtr = 197,

Bcai = 49,

Bcajsr = 53,

Bcass = 51,

ENa = 56,

EK = 57,

EKs = 58,

mss = 0,

tm = 13,

hss = 1,

thf = 14,

ths = 15,

h = 59,

jss = 16,

tj = 30,

hssp = 31,

thsp = 40,

hp = 60,

tjp = 41,

fINap = 61,

mLss = 32,

tmL = 42,

hLss = 2,

hLssp = 3,

fINaLp = 63,

ass = 4,

ta = 17,

iss = 5,

delta\_epi = 18,

tiF\_b = 33,

tiS\_b = 43,

tiF = 46,

tiS = 48,

AiF = 65,

AiS = 66,

i = 67,

assp = 34,

dti\_develop = 50,

dti\_recover = 52,

tiFp = 54,

tiSp = 55,

ip = 68,

fItop = 69,

dss = 6,

fss = 7,

f = 71,

fcass = 19,

Afcaf = 72,

Afcas = 73,

fca = 74,

fp = 75,

fcap = 76,

km2n = 8,

anca = 20,

PhiCaL = 77,

PhiCaNa = 78,

PhiCaK = 79,

fICaLp = 80,

td = 21,

tff = 22,

tfs = 23,

tfcaf = 35,

tfcas = 36,

tffp = 37,

tfcafp = 44,

xrss = 9,

txrf = 24,

txrs = 25,

Axrf = 88,

Axrs = 91,

xr = 94,

rkr = 95,

xs1ss = 10,

xs2ss = 26,

txs1 = 27,

KsCa = 97,

txs2 = 38,

xk1ss = 11,

txk1 = 28,

rk1 = 99,

hna = 102,

hca = 101,

h1\_i = 103,

h2\_i = 104,

h3\_i = 105,

h4\_i = 106,

h5\_i = 107,

h6\_i = 108,

h7\_i = 109,

h8\_i = 110,

h9\_i = 111,

k3p\_i = 112,

k3pp\_i = 113,

k3\_i = 114,

k4\_i = 117,

k4p\_i = 115,

k4pp\_i = 116,

k6\_i = 118,

k7\_i = 119,

k8\_i = 120,

x1\_i = 121,

x2\_i = 122,

x3\_i = 123,

x4\_i = 124,

E1\_i = 125,

E2\_i = 126,

E3\_i = 127,

E4\_i = 128,

allo\_i = 129,

JncxNa\_i = 130,

JncxCa\_i = 131,

h1\_ss = 133,

h2\_ss = 134,

h3\_ss = 135,

h4\_ss = 136,

h5\_ss = 137,

h6\_ss = 138,

h7\_ss = 139,

h8\_ss = 140,

h9\_ss = 141,

k3p\_ss = 142,

k3pp\_ss = 143,

k3\_ss = 144,

k4\_ss = 147,

k4p\_ss = 145,

k4pp\_ss = 146,

k6\_ss = 148,

k7\_ss = 149,

k8\_ss = 150,

x1\_ss = 151,

x2\_ss = 152,

x3\_ss = 153,

x4\_ss = 154,

E1\_ss = 155,

E2\_ss = 156,

E3\_ss = 157,

E4\_ss = 158,

allo\_ss = 159,

JncxNa\_ss = 160,

JncxCa\_ss = 161,

Knai = 163,

Knao = 164,

P = 165,

a1 = 166,

b2 = 167,

a3 = 168,

b3 = 169,

b4 = 170,

x1 = 171,

x2 = 172,

x3 = 173,

x4 = 174,

E1 = 175,

E2 = 176,

E3 = 177,

E4 = 178,

JnakNa = 179,

JnakK = 180,

xkb = 182,

Jrel\_inf = 86,

tau\_rel = 92,

Jrel\_infp = 87,

Jrel\_temp = 84,

tau\_relp = 93,

Jrel\_inf\_temp = 83,

fJrelp = 190,

tau\_rel\_temp = 89,

tau\_relp\_temp = 90,

Jupnp = 192,

Jupp = 193,

fJupp = 194,

Jleak = 195,

step\_low = 196,

step\_high = 197,

};

enum E\_CONSTANTS\_T{

celltype = 0,

nao = 1,

cao = 2,

ko = 3,

R = 4,

T = 5,

F = 6,

zna = 7,

zca = 8,

zk = 9,

L = 10,

rad = 11,

vcell = 104,

Ageo = 113,

Acap = 117,

vmyo = 118,

vnsr = 119,

vjsr = 120,

vss = 121,

amp = 12,

duration = 13,

KmCaMK = 14,

aCaMK = 15,

bCaMK = 16,

CaMKo = 17,

KmCaM = 18,

cmdnmax\_b = 19,

cmdnmax = 93,

kmcmdn = 20,

trpnmax = 21,

kmtrpn = 22,

BSRmax = 23,

KmBSR = 24,

BSLmax = 25,

KmBSL = 26,

csqnmax = 27,

kmcsqn = 28,

cm = 29,

PKNa = 30,

mssV1 = 31,

mssV2 = 32,

mtV1 = 33,

mtV2 = 34,

mtD1 = 35,

mtD2 = 36,

mtV3 = 37,

mtV4 = 38,

hssV1 = 39,

hssV2 = 40,

Ahs = 94,

Ahf = 41,

GNa = 42,

thL = 43,

thLp = 95,

GNaL\_b = 44,

GNaL = 96,

Gto\_b = 45,

Gto = 97,

Kmn = 46,

k2n = 47,

PCa\_b = 48,

Aff = 98,

Afs = 109,

PCa = 99,

PCap = 110,

PCaNa = 111,

PCaK = 112,

PCaNap = 115,

PCaKp = 116,

tjca = 100,

GKr\_b = 49,

GKr = 101,

GKs\_b = 50,

GKs = 102,

GK1 = 103,

GK1\_b = 51,

kna1 = 52,

kna2 = 53,

kna3 = 54,

kasymm = 55,

wna = 56,

wca = 57,

wnaca = 58,

kcaon = 59,

kcaoff = 60,

qna = 61,

qca = 62,

KmCaAct = 63,

Gncx\_b = 64,

Gncx = 128,

h10\_i = 122,

h11\_i = 123,

h12\_i = 124,

k1\_i = 125,

k2\_i = 126,

k5\_i = 127,

h10\_ss = 129,

h11\_ss = 130,

h12\_ss = 131,

k1\_ss = 132,

k2\_ss = 133,

k5\_ss = 134,

k1p = 65,

k1m = 66,

k2p = 67,

k2m = 68,

k3p = 69,

k3m = 70,

k4p = 71,

k4m = 72,

Knai0 = 73,

Knao0 = 74,

delta = 75,

Kki = 76,

Kko = 77,

MgADP = 78,

MgATP = 79,

Kmgatp = 80,

H = 81,

eP = 82,

Khp = 83,

Knap = 84,

Kxkur = 85,

Pnak\_b = 86,

Pnak = 138,

b1 = 135,

a2 = 136,

a4 = 137,

GKb\_b = 87,

GKb = 105,

PNab = 88,

PCab = 89,

GpCa = 90,

KmCap = 91,

bt = 92,

a\_rel = 106,

btp = 107,

a\_relp = 114,

upScale = 108,

stim\_start = 139,

BCL = 140,

//stim\_end = 141,

// Added by ALI

// Additional constants for cvar

Jrel\_scale = 141,

Jup\_scale = 142,

Jtr\_scale = 143,

Jleak\_scale = 144,

//KCaMK\_scale = 144,

};

enum E\_STATES\_T{

V = 0,

CaMKt = 1,

cass = 2,

nai = 3,

nass = 4,

ki = 5,

kss = 6,

cansr = 7,

cajsr = 8,

cai = 9,

m = 10,

hf = 11,

hs = 12,

j = 13,

hsp = 14,

jp = 15,

mL = 16,

hL = 17,

hLp = 18,

a = 19,

iF = 20,

iS = 21,

ap = 22,

iFp = 23,

iSp = 24,

d = 25,

ff = 26,

fs = 27,

fcaf = 28,

fcas = 29,

jca = 30,

ffp = 31,

fcafp = 32,

nca = 33,

xrf = 34,

xrs = 35,

xs1 = 36,

xs2 = 37,

xk1 = 38,

Jrelnp = 39,

Jrelp = 40,

};

#endif

## ‘modules’ Folder

The modules folder is a vital component of the project, containing some essential files that implement the core functionalities of the simulation. These files are modularised into source (.cpp or .cu) and header (.hpp or .cuh) formats, ensuring that the codebase is both organised and extensible. Each module serves a specific purpose, contributing to the overall framework of the simulation pipeline. As in the writing of this thesis, we found that some scripts were redundant and will be removed in the next updates.

## cipa\_t.cu and cipa\_t.cuh

These files handle data type related to the formatting of CiPA (Comprehensive in vitro Proarrhythmia Assay) metrics, which are critical and become standard for assessing drug-induced proarrhythmia risks. All implementation in cipa\_t.cu has been migrated to its header file (cipa\_t.cuh) and will be removed in the next updates. Header file of cipa\_t.cuh contains commented old vectors and naturalised 1D arrays as such:

#ifndef CIPA\_T\_HPP

#define CIPA\_T\_HPP

#include <map>

#include <string>

#include <cuda\_runtime.h>

// using std::multimap;

// using std::string;

\_\_global\_\_ struct cipa\_t{

double qnet\_ap;

double qnet4\_ap;

double inal\_auc\_ap;

double ical\_auc\_ap;

double qnet\_cl;

double qnet4\_cl;

double inal\_auc\_cl;

double ical\_auc\_cl;

double dvmdt\_repol;

double vm\_peak;

double vm\_valley;

// multimap<double, double> vm\_data;

// multimap<double, double> dvmdt\_data;

// multimap<double, double> cai\_data;

// multimap<double, string> ires\_data;

// multimap<double, string> inet\_data;

// multimap<double, string> qnet\_data;

// multimap<double, string> inet4\_data;

// multimap<double, string> qnet4\_data;

// multimap<double, string> time\_series\_data;

// temporary fix for this

double vm\_data[7000];

double vm\_time[7000];

double dvmdt\_data[7000];

double dvmdt\_time[7000];

double cai\_data[7000];

double cai\_time[7000];

double ires\_data[7000];

double ires\_time[7000];

double inet\_data[7000];

double inet\_time[7000];

double qnet\_data[7000];

double qnet\_time[7000];

double inet4\_data[7000];

double inet4\_time[7000];

double qnet4\_data[7000];

double qnet4\_time[7000];

// double time\_series\_data[7000];

// double time\_series\_time[7000];

// \_\_device\_\_ cipa\_t();

// \_\_device\_\_ cipa\_t( const cipa\_t &source );

// cipa\_t& operator=(const cipa\_t & source);

// \_\_device\_\_ void copy(const cipa\_t &source);

// \_\_device\_\_ void init(const double vm\_val);

// \_\_device\_\_ void clear\_time\_result();

};

#endif

## drug\_conc.cpp and drug\_conc.hpp

These files manage the drug concentration data used in the simulations. These act as utility file to lookup if user did not declare the concentration value and drugs used is in the scope of CiPA. Lookup process is happening in the CPU domain. Below is the list of known drug by this utility function and its implementation using map:

#include "drug\_conc.hpp"

float getValue(const std::unordered\_map<std::string, float>& drugConc, const std::string& key, float defaultValue) {

auto it = drugConc.find(key);

if (it != drugConc.end()) {

return it->second;

}

return defaultValue;

}

// Instantiate the dictionary with keys and values

std::unordered\_map<std::string, float> drugConcentration = {

{"azimilide", 70.0f},

{"bepridil", 33.0f},

{"disopyramide", 742.0f},

{"dofetilide", 2.0f},

{"ibutilide", 100.0f},

{"quinidine", 3237.0f},

{"sotalol", 14690.0f},

{"vandetanib", 255.0f},

{"astemizole", 0.26f},

{"chlorpromazine", 38.0f},

{"cisapride", 2.6f},

{"clarithromycin", 1206.0f},

{"clozapine", 71.0f},

{"domperidone", 19.0f},

{"droperidol", 6.3f},

{"ondansetron", 139.0f},

{"pimozide", 0.431f},

{"risperidone", 1.81f},

{"terfenadine", 4.0f},

{"diltiazem", 122.0f},

{"loratadine", 0.45f},

{"metoprolol", 1800.0f},

{"mexiletine", 4129.0f},

{"nifedipine", 7.7f},

{"nitrendipine", 3.02f},

{"ranolazine", 1948.2f},

{"tamoxifen", 21.0f},

{"verapamil", 81.0f}

};

And header file:

#ifndef DRUG\_CONC\_HPP

#define DRUG\_CONC\_HPP

#include <unordered\_map>

#include <string>

// Declare the dictionary function template

float getValue(const std::unordered\_map<std::string, float>& drugConc, const std::string& key, float defaultValue = 0.0);

// Declare the dictionary extern

extern std::unordered\_map<std::string, float> drugConcentration;

#endif // DRUG\_CONC\_HPP

## glob\_funct.cpp and glob\_funct.hpp

Global function or glob\_funct encapsulate global utility functions that are used across various parts of the codebase. At the time this thesis was written, this script regulates the reading of flags and input deck. There are some unused functions that will be removed in the future. The global function script is implemented as below:

#include "glob\_funct.hpp"

// #include "../libs/zip.h"

#include <cstdarg>

#include <cstdio>

#include <cstdlib>

#include <cstring>

// to make it more "portable" between OSes.

#if defined \_WIN32

#include <direct.h>

#define snprintf \_snprintf

#define vsnprintf \_vsnprintf

#define strcasecmp \_stricmp

#define strncasecmp \_strnicmp

#else

#include <dirent.h>

#endif

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

void mpi\_printf(unsigned short node\_id, const char \*fmt, ...)

{

va\_list args;

va\_start(args, fmt);

vprintf(fmt, args);

va\_end(args);

}

void mpi\_fprintf(unsigned short node\_id, FILE \*stream, const char \*fmt, ...)

{

#ifndef \_WIN32

if(mympi::rank == node\_id){

va\_list args;

va\_start(args, fmt);

vfprintf(stream, fmt, args);

va\_end(args);

}

#else

va\_list args;

va\_start(args, fmt);

vfprintf(stream, fmt, args);

va\_end(args);

#endif

}

void edison\_assign\_params(int argc, char \*argv[], param\_t \*p\_param)

{

bool is\_default;

char buffer[100];

char key[100];

char value[100];

char file\_name[150];

FILE \*fp\_inputdeck;

// parameters from arguments

for (int idx = 1; idx < argc; idx += 2) {

if (!strcasecmp(argv[idx], "-input\_deck"))

strncpy(file\_name, argv[idx + 1], sizeof(file\_name));

else if (!strcasecmp(argv[idx], "-hill\_file"))

strncpy(p\_param->hill\_file, argv[idx + 1], sizeof(p\_param->hill\_file));

else if (!strcasecmp(argv[idx], "-cvar\_file"))

strncpy(p\_param->cvar\_file, argv[idx + 1], sizeof(p\_param->cvar\_file));

}

is\_default = false;

fp\_inputdeck = fopen( file\_name, "r");

if(fp\_inputdeck == NULL){

fprintf(stderr, "Cannot open input deck file %s!!!\nUse default value as the failsafe.\n", file\_name);

is\_default = true;

}

// read input\_deck line by line

// and store each line to the buffer

while ( is\_default == false && fgets( buffer, 100, fp\_inputdeck ) != NULL ) {

sscanf( buffer, "%s %\*s %s", key, value );

if (strcasecmp(key, "Simulation\_Mode") == 0) {

p\_param->simulation\_mode = strtod( value, NULL );

}

else if (strcasecmp(key, "Celltype") == 0) {

p\_param->celltype = strtod( value, NULL );

}

else if (strcasecmp(key, "Is\_Dutta") == 0) {

p\_param->is\_dutta = strtol( value, NULL, 10 );

}

else if (strcasecmp(key, "Use\_Conductance\_Variability") == 0) {

p\_param->is\_cvar = strtol( value, NULL, 10 );

}

else if (strcasecmp(key, "Pace\_Find\_Steepest") == 0) {

p\_param->find\_steepest\_start = strtod( value, NULL);

}

else if (strcasecmp(key, "GPU\_Index") == 0) {

p\_param->gpu\_index = strtod( value, NULL);

}

else if (strcasecmp(key, "Basic\_Cycle\_Length") == 0) {

p\_param->bcl = strtod( value, NULL );

}

else if (strcasecmp(key, "Number\_of\_Pacing") == 0) {

p\_param->pace\_max = strtod( value, NULL );

}

else if (strcasecmp(key, "Time\_Step") == 0) {

p\_param->dt = strtod( value, NULL );

}

//TODO: #Automation 1. eliminate drug\_name and concentration on drug\_name

// else if (strcasecmp(key, "Drug\_Name") == 0) {

// strncpy( p\_param->drug\_name, value, sizeof(p\_param->concs) );

// }

// else if (strcasecmp(key, "Concentrations") == 0) {

// p\_param->conc = strtod( value, NULL );

// }

}

if( is\_default == false ) fclose( fp\_inputdeck );

}

int make\_directory(const char\* dirname )

{

#if defined \_WIN32

return \_mkdir(dirname);

#else

return mkdir(dirname, 0775);

#endif

}

int is\_file\_existed(const char\* pathname)

{

#if defined \_WIN32

struct \_stat buf;

return \_stat( pathname, &buf );

#else

struct stat st = {0};

return stat(pathname, &st);

#endif

}

## glob\_type.cpp and glob\_type.hpp

The glob\_type files define and manage the global data types and structures used throughout the project. For now, this script is being used to store IC50 data. The glob\_type.hpp is found unused and will be removed in the next update. As glob\_type.cpp is still being used, here is our implementation into it:

#ifndef GLOB\_TYPE\_HPP

#define GLOB\_TYPE\_HPP

#include <vector>

// global variable for MPI.

struct mympi

{

static char host\_name[255];

static int host\_name\_len;

static int rank;

static int size;

};

// data structure for IC50

typedef struct row\_data { double data[14]; } row\_data;

typedef std::vector< row\_data > drug\_t;

// data structure to store

// ICaL/INaL control value

// for calculating qinward

// control means 0 concentration

// otherwise, drugs

typedef struct{

double ical\_auc\_control;

double inal\_auc\_control;

double ical\_auc\_drug;

double inal\_auc\_drug;

} qinward\_t;

#endif

## gpu.cu and gpu.cuh

The gpu files represent the core computational engine of the simulation, leveraging the parallel processing capabilities of NVIDIA GPUs to achieve significant speedups in simulation tasks. These files are essential for the framework's computational efficiency, enabling the processing of thousands of samples in parallel.

gpu.cu is the implementation file that contains CUDA-specific kernels and device functions. CUDA kernels are at the heart of the GPU simulation, responsible for solving the differential equations of the cell models, processing drug interactions, and performing large-scale computations for multiple samples concurrently. The kernels in this file are designed to optimise memory usage and minimise latency, ensuring the efficient execution of simulations. Header gpu.cuh serves as the header file that defines the interface for the GPU-specific functionalities implemented in gpu.cu. It declares the prototypes of CUDA kernels and device functions, providing a structured entry point for other parts of the code to interact with GPU-based operations. The main gpu.cu code will look as this:

#include "../cellmodels/Ohara\_Rudy\_2011.hpp"

#include <stdio.h>

#include <cuda\_runtime.h>

#include <cuda.h>

#include "glob\_funct.hpp"

#include "glob\_type.hpp"

#include "gpu\_glob\_type.cuh"

#include "gpu.cuh"

/\*

all kernel function has been moved. Unlike the previous GPU code, now we seperate everything into each modules.

all modules here are optimised for GPU and slightly different than the original component based code

differences are related to GPU offset calculations

\*/

\_\_device\_\_ void kernel\_DoDrugSim(double \*d\_ic50, double \*d\_cvar, double d\_conc, double \*d\_CONSTANTS, double \*d\_STATES, double \*d\_RATES, double \*d\_ALGEBRAIC,

double \*d\_STATES\_RESULT,

// double \*time, double \*states, double \*out\_dt, double \*cai\_result,

// double \*ina, double \*inal,

// double \*ical, double \*ito,

// double \*ikr, double \*iks,

// double \*ik1,

double \*tcurr, double \*dt, unsigned short sample\_id, unsigned int sample\_size,

cipa\_t \*temp\_result, cipa\_t \*cipa\_result,

param\_t \*p\_param

)

{

unsigned int input\_counter = 0;

int num\_of\_constants = 145;

int num\_of\_states = 41;

int num\_of\_algebraic = 199;

int num\_of\_rates = 41;

// INIT STARTS

temp\_result[sample\_id].qnet\_ap = 0.;

temp\_result[sample\_id].qnet4\_ap = 0.;

temp\_result[sample\_id].inal\_auc\_ap = 0.;

temp\_result[sample\_id].ical\_auc\_ap = 0.;

temp\_result[sample\_id].qnet\_cl = 0.;

temp\_result[sample\_id].qnet4\_cl = 0.;

temp\_result[sample\_id].inal\_auc\_cl = 0.;

temp\_result[sample\_id].ical\_auc\_cl = 0.;

temp\_result[sample\_id].dvmdt\_repol = -999;

temp\_result[sample\_id].vm\_peak = -999;

temp\_result[sample\_id].vm\_valley = d\_STATES[(sample\_id \* num\_of\_states) +V];

cipa\_result[sample\_id].qnet\_ap = 0.;

cipa\_result[sample\_id].qnet4\_ap = 0.;

cipa\_result[sample\_id].inal\_auc\_ap = 0.;

cipa\_result[sample\_id].ical\_auc\_ap = 0.;

cipa\_result[sample\_id].qnet\_cl = 0.;

cipa\_result[sample\_id].qnet4\_cl = 0.;

cipa\_result[sample\_id].inal\_auc\_cl = 0.;

cipa\_result[sample\_id].ical\_auc\_cl = 0.;

cipa\_result[sample\_id].dvmdt\_repol = -999;

cipa\_result[sample\_id].vm\_peak = -999;

cipa\_result[sample\_id].vm\_valley = d\_STATES[(sample\_id \* num\_of\_states) +V];

// INIT ENDS

bool is\_peak = false;

// to search max dvmdt repol

tcurr[sample\_id] = 0.0;

dt[sample\_id] = p\_param->dt;

double tmax;

double max\_time\_step = 1.0, time\_point = 25.0;

double dt\_set;

int cipa\_datapoint = 0;

const double bcl = p\_param->bcl;

// const double inet\_vm\_threshold = p\_param->inet\_vm\_threshold;

// const unsigned short pace\_max = 300;

// const unsigned short pace\_max = 1000;

const unsigned short pace\_max = p\_param->pace\_max;

const unsigned short last\_drug\_check\_pace = p\_param->find\_steepest\_start;

double conc = d\_conc; //mmol

double type = p\_param->celltype;

double epsilon = 10E-14;

// eligible AP shape means the Vm\_peak > 0.

bool is\_eligible\_AP;

// Vm value at 30% repol, 50% repol, and 90% repol, respectively.

double vm\_repol30, vm\_repol50, vm\_repol90;

double t\_peak\_capture = 0.0;

unsigned short pace\_steepest = 0;

bool init\_states\_captured = false;

// qnet\_ap/inet\_ap values

double inet\_ap, qnet\_ap, inet4\_ap, qnet4\_ap, inet\_cl, qnet\_cl, inet4\_cl, qnet4\_cl;

double inal\_auc\_ap, ical\_auc\_ap,inal\_auc\_cl, ical\_auc\_cl;

// qinward\_cl;

// char buffer[255];

// static const int CALCIUM\_SCALING = 1000000;

// static const int CURRENT\_SCALING = 1000;

// printf("Core %d:\n",sample\_id);

initConsts(d\_CONSTANTS, d\_STATES, type, conc, d\_ic50, d\_cvar, p\_param->is\_dutta, p\_param->is\_cvar, sample\_id);

applyDrugEffect(d\_CONSTANTS, conc, d\_ic50, epsilon, sample\_id);

d\_CONSTANTS[BCL + (sample\_id \* num\_of\_constants)] = bcl;

// generate file for time-series output

tmax = pace\_max \* bcl;

int pace\_count = 0;

// printf("%d,%lf,%lf,%lf,%lf\n", sample\_id, dt[sample\_id], tcurr[sample\_id], d\_STATES[V + (sample\_id \* num\_of\_states)],d\_RATES[V + (sample\_id \* num\_of\_rates)]);

// printf("%lf,%lf,%lf,%lf,%lf\n", d\_ic50[0 + (14\*sample\_id)], d\_ic50[1+ (14\*sample\_id)], d\_ic50[2+ (14\*sample\_id)], d\_ic50[3+ (14\*sample\_id)], d\_ic50[4+ (14\*sample\_id)]);

while (tcurr[sample\_id]<tmax)

{

computeRates(tcurr[sample\_id], d\_CONSTANTS, d\_RATES, d\_STATES, d\_ALGEBRAIC, sample\_id);

dt\_set = set\_time\_step( tcurr[sample\_id], time\_point, max\_time\_step,

d\_CONSTANTS,

d\_RATES,

d\_STATES,

d\_ALGEBRAIC,

sample\_id);

//euler only

// dt\_set = 0.005;

// printf("tcurr at core %d: %lf\n",sample\_id,tcurr[sample\_id]);

if (floor((tcurr[sample\_id] + dt\_set) / bcl) == floor(tcurr[sample\_id] / bcl)) {

dt[sample\_id] = dt\_set;

// printf("dt : %lf\n",dt\_set);

// it goes in here, but it does not, you know, adds the pace,

}

else{

dt[sample\_id] = (floor(tcurr[sample\_id] / bcl) + 1) \* bcl - tcurr[sample\_id];

// new part starts

if( is\_eligible\_AP && pace\_count >= pace\_max-last\_drug\_check\_pace) {

temp\_result[sample\_id].qnet\_ap = qnet\_ap;

temp\_result[sample\_id].qnet4\_ap = qnet4\_ap;

temp\_result[sample\_id].inal\_auc\_ap = inal\_auc\_ap;

temp\_result[sample\_id].ical\_auc\_ap = ical\_auc\_ap;

temp\_result[sample\_id].qnet\_cl = qnet\_cl;

temp\_result[sample\_id].qnet4\_cl = qnet4\_cl;

temp\_result[sample\_id].inal\_auc\_cl = inal\_auc\_cl;

temp\_result[sample\_id].ical\_auc\_cl = ical\_auc\_cl;

// fprintf(fp\_vmdebug, "%hu,%.2lf,%.2lf,%.2lf,%.2lf,%.2lf,%.2lf\n", pace\_count,t\_peak\_capture,temp\_result.vm\_peak,vm\_repol30,vm\_repol50,vm\_repol90,temp\_result.dvmdt\_repol);

// replace result with steeper repolarization AP or first pace from the last 250 paces

// if( temp\_result->dvmdt\_repol > cipa\_result.dvmdt\_repol ) {

// pace\_steepest = pace\_count;

// cipa\_result = temp\_result;

// }

if( temp\_result[sample\_id].dvmdt\_repol > cipa\_result[sample\_id].dvmdt\_repol ) {

pace\_steepest = pace\_count;

// printf("Steepest pace updated: %d dvmdt\_repol: %lf\n",pace\_steepest,temp\_result[sample\_id].dvmdt\_repol);

// cipa\_result = temp\_result;

cipa\_result[sample\_id].qnet\_ap = temp\_result[sample\_id].qnet\_ap;

cipa\_result[sample\_id].qnet4\_ap = temp\_result[sample\_id].qnet4\_ap;

cipa\_result[sample\_id].inal\_auc\_ap = temp\_result[sample\_id].inal\_auc\_ap;

cipa\_result[sample\_id].ical\_auc\_ap = temp\_result[sample\_id].ical\_auc\_ap;

cipa\_result[sample\_id].qnet\_cl = temp\_result[sample\_id].qnet\_cl;

cipa\_result[sample\_id].qnet4\_cl = temp\_result[sample\_id].qnet4\_cl;

cipa\_result[sample\_id].inal\_auc\_cl = temp\_result[sample\_id].inal\_auc\_cl;

cipa\_result[sample\_id].ical\_auc\_cl = temp\_result[sample\_id].ical\_auc\_cl;

cipa\_result[sample\_id].dvmdt\_repol = temp\_result[sample\_id].dvmdt\_repol;

cipa\_result[sample\_id].vm\_peak = temp\_result[sample\_id].vm\_peak;

cipa\_result[sample\_id].vm\_valley = d\_STATES[(sample\_id \* num\_of\_states) +V];

is\_peak = true;

init\_states\_captured = false;

}

else{

is\_peak = false;

}

};

inet\_ap = 0.;

qnet\_ap = 0.;

inet4\_ap = 0.;

qnet4\_ap = 0.;

inal\_auc\_ap = 0.;

ical\_auc\_ap = 0.;

inet\_cl = 0.;

qnet\_cl = 0.;

inet4\_cl = 0.;

qnet4\_cl = 0.;

inal\_auc\_cl = 0.;

ical\_auc\_cl = 0.;

t\_peak\_capture = 0.;

// temp\_result->init( p\_cell->STATES[V]);

temp\_result[sample\_id].qnet\_ap = 0.;

temp\_result[sample\_id].qnet4\_ap = 0.;

temp\_result[sample\_id].inal\_auc\_ap = 0.;

temp\_result[sample\_id].ical\_auc\_ap = 0.;

temp\_result[sample\_id].qnet\_cl = 0.;

temp\_result[sample\_id].qnet4\_cl = 0.;

temp\_result[sample\_id].inal\_auc\_cl = 0.;

temp\_result[sample\_id].ical\_auc\_cl = 0.;

temp\_result[sample\_id].dvmdt\_repol = -999;

temp\_result[sample\_id].vm\_peak = -999;

temp\_result[sample\_id].vm\_valley = d\_STATES[(sample\_id \* num\_of\_states) +V];

// end of init

pace\_count++;

input\_counter = 0; // at first, we reset the input counter since we re gonna only take one, but I remember we don't have this kind of thing previously, so do we need this still?

cipa\_datapoint = 0; // new pace? reset variables related to saving the values,

is\_eligible\_AP = false;

// new part ends

if(sample\_id == 0 || (sample\_id % 1000 == 0 && sample\_id>999)){

printf("core: %d pace count: %d, steepest: %d, dvmdt\_repol: %lf, conc: %lf, celltype: %lf\n",sample\_id,pace\_count, pace\_steepest, cipa\_result[sample\_id].dvmdt\_repol, conc, d\_CONSTANTS[(sample\_id \* num\_of\_constants) + celltype]);

}

// printf("core: %d pace count: %d t: %lf, steepest: %d, dvmdt\_repol: %lf, t\_peak: %lf\n",sample\_id,pace\_count, tcurr[sample\_id], pace\_steepest, cipa\_result[sample\_id].dvmdt\_repol,t\_peak\_capture);

// writen = false;

}

solveAnalytical(d\_CONSTANTS, d\_STATES, d\_ALGEBRAIC, d\_RATES, dt[sample\_id], sample\_id);

// solveEuler(d\_STATES, d\_RATES, dt[sample\_id], sample\_id);

if (pace\_count >= pace\_max-last\_drug\_check\_pace)

{

if( tcurr[sample\_id] > ((d\_CONSTANTS[(sample\_id \* num\_of\_constants) +BCL]\*pace\_count)+(d\_CONSTANTS[(sample\_id \* num\_of\_constants) +stim\_start]+2)) &&

tcurr[sample\_id] < ((d\_CONSTANTS[(sample\_id \* num\_of\_constants) +BCL]\*pace\_count)+(d\_CONSTANTS[(sample\_id \* num\_of\_constants) +stim\_start]+10)) &&

abs(d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INa]) < 1)

{

// printf("check 1\n");

if( d\_STATES[(sample\_id \* num\_of\_states) +V] > temp\_result[sample\_id].vm\_peak )

{

temp\_result[sample\_id].vm\_peak = d\_STATES[(sample\_id \* num\_of\_states) +V];

if(temp\_result[sample\_id].vm\_peak > 0)

{

vm\_repol30 = temp\_result[sample\_id].vm\_peak - (0.3 \* (temp\_result[sample\_id].vm\_peak - temp\_result[sample\_id].vm\_valley));

vm\_repol50 = temp\_result[sample\_id].vm\_peak - (0.5 \* (temp\_result[sample\_id].vm\_peak - temp\_result[sample\_id].vm\_valley));

vm\_repol90 = temp\_result[sample\_id].vm\_peak - (0.9 \* (temp\_result[sample\_id].vm\_peak - temp\_result[sample\_id].vm\_valley));

is\_eligible\_AP = true;

t\_peak\_capture = tcurr[sample\_id];

}

else is\_eligible\_AP = false;

}

}

else if( tcurr[sample\_id] > ((d\_CONSTANTS[(sample\_id \* num\_of\_constants) +BCL]\*pace\_count)+(d\_CONSTANTS[(sample\_id \* num\_of\_constants) +stim\_start]+10)) && is\_eligible\_AP )

{

if( d\_RATES[(sample\_id \* num\_of\_rates) +V] > temp\_result[sample\_id].dvmdt\_repol &&

d\_STATES[(sample\_id \* num\_of\_states) +V] <= vm\_repol30 &&

d\_STATES[(sample\_id \* num\_of\_states) +V] >= vm\_repol90 )

{

temp\_result[sample\_id].dvmdt\_repol = d\_RATES[(sample\_id \* num\_of\_rates) +V];

// printf("check 4\n");

}

}

// calculate AP shape

if(is\_eligible\_AP && d\_STATES[(sample\_id \* num\_of\_states) +V] > vm\_repol90)

{

inet\_ap = (d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INaL]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +ICaL]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +Ito]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +IKr]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +IKs]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +IK1]);

inet4\_ap = (d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INaL]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +ICaL]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +IKr]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INa]);

qnet\_ap += (inet\_ap \* dt[sample\_id])/1000.;

qnet4\_ap += (inet4\_ap \* dt[sample\_id])/1000.;

inal\_auc\_ap += (d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INaL]\*dt[sample\_id]);

ical\_auc\_ap += (d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +ICaL]\*dt[sample\_id]);

}

inet\_cl = (d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INaL]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +ICaL]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +Ito]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +IKr]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +IKs]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +IK1]);

inet4\_cl = (d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INaL]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +ICaL]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +IKr]+d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INa]);

qnet\_cl += (inet\_cl \* dt[sample\_id])/1000.;

qnet4\_cl += (inet4\_cl \* dt[sample\_id])/1000.;

inal\_auc\_cl += (d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +INaL]\*dt[sample\_id]);

ical\_auc\_cl += (d\_ALGEBRAIC[(sample\_id \* num\_of\_algebraic) +ICaL]\*dt[sample\_id]);

// save temporary result -> ALL TEMP RESULTS IN, TEMP RESULT != WRITTEN RESULT

if((pace\_count >= pace\_max-last\_drug\_check\_pace) && (is\_peak == true) && (pace\_count<pace\_max) )

{

// printf("input\_counter: %d\n",input\_counter);

// datapoint\_at\_this\_moment = tcurr[sample\_id] - (pace\_count \* bcl);

temp\_result[sample\_id].cai\_data[cipa\_datapoint] = d\_STATES[(sample\_id \* num\_of\_states) +cai] ;

temp\_result[sample\_id].cai\_time[cipa\_datapoint] = tcurr[sample\_id];

temp\_result[sample\_id].vm\_data[cipa\_datapoint] = d\_STATES[(sample\_id \* num\_of\_states) +V];

temp\_result[sample\_id].vm\_time[cipa\_datapoint] = tcurr[sample\_id];

temp\_result[sample\_id].dvmdt\_data[cipa\_datapoint] = d\_RATES[(sample\_id \* num\_of\_rates) +V];

temp\_result[sample\_id].dvmdt\_time[cipa\_datapoint] = tcurr[sample\_id];

if(init\_states\_captured == false){

for(int counter=0; counter<num\_of\_states; counter++){

d\_STATES\_RESULT[(sample\_id \* num\_of\_states) + counter] = d\_STATES[(sample\_id \* num\_of\_states) + counter];

}

init\_states\_captured = true;

}

// time series result

// time[input\_counter + sample\_id] = tcurr[sample\_id];

// states[input\_counter + sample\_id] = d\_STATES[V + (sample\_id \* num\_of\_states)];

// out\_dt[input\_counter + sample\_id] = d\_RATES[V + (sample\_id \* num\_of\_states)];

// cai\_result[input\_counter + sample\_id] = d\_ALGEBRAIC[cai + (sample\_id \* num\_of\_algebraic)];

// ina[input\_counter + sample\_id] = d\_ALGEBRAIC[INa + (sample\_id \* num\_of\_algebraic)] ;

// inal[input\_counter + sample\_id] = d\_ALGEBRAIC[INaL + (sample\_id \* num\_of\_algebraic)] ;

// ical[input\_counter + sample\_id] = d\_ALGEBRAIC[ICaL + (sample\_id \* num\_of\_algebraic)] ;

// ito[input\_counter + sample\_id] = d\_ALGEBRAIC[Ito + (sample\_id \* num\_of\_algebraic)] ;

// ikr[input\_counter + sample\_id] = d\_ALGEBRAIC[IKr + (sample\_id \* num\_of\_algebraic)] ;

// iks[input\_counter + sample\_id] = d\_ALGEBRAIC[IKs + (sample\_id \* num\_of\_algebraic)] ;

// ik1[input\_counter + sample\_id] = d\_ALGEBRAIC[IK1 + (sample\_id \* num\_of\_algebraic)] ;

input\_counter = input\_counter + sample\_size;

cipa\_datapoint = cipa\_datapoint + 1; // this causes the resource usage got so mega and crashed in running

} // temporary guard ends here

} // end the last 250 pace operations

tcurr[sample\_id] = tcurr[sample\_id] + dt[sample\_id];

//printf("t after addition: %lf\n", tcurr[sample\_id]);

} // while loop ends here

// \_\_syncthreads();

}

\_\_global\_\_ void kernel\_DrugSimulation(double \*d\_ic50, double \*d\_cvar, double \*d\_conc, double \*d\_CONSTANTS, double \*d\_STATES, double \*d\_RATES, double \*d\_ALGEBRAIC,

double \*d\_STATES\_RESULT,

// double \*time, double \*states, double \*out\_dt, double \*cai\_result,

// double \*ina, double \*inal,

// double \*ical, double \*ito,

// double \*ikr, double \*iks,

// double \*ik1,

unsigned int sample\_size,

cipa\_t \*temp\_result, cipa\_t \*cipa\_result,

param\_t \*p\_param

)

{

unsigned short thread\_id;

thread\_id = blockIdx.x \* blockDim.x + threadIdx.x;

if (thread\_id >= sample\_size) return;

double time\_for\_each\_sample[10000];

double dt\_for\_each\_sample[10000];

// cipa\_t temp\_per\_sample[2000];

// cipa\_t cipa\_per\_sample[2000];

// printf("in\n");

// printf("Calculating %d\n",thread\_id);

kernel\_DoDrugSim(d\_ic50, d\_cvar, d\_conc[thread\_id], d\_CONSTANTS, d\_STATES, d\_RATES, d\_ALGEBRAIC,

d\_STATES\_RESULT,

// time, states, out\_dt, cai\_result,

// ina, inal,

// ical, ito,

// ikr, iks,

// ik1,

time\_for\_each\_sample, dt\_for\_each\_sample, thread\_id, sample\_size,

temp\_result, cipa\_result,

p\_param

);

// \_\_syncthreads();

// printf("Calculation for core %d done\n",sample\_id);

}

The header file of gpu.hpp declared as follows:

#ifndef GPU\_CUH

#define GPU\_CUH

#include <cuda\_runtime.h>

#include <cuda.h>

#include "cipa\_t.cu"

\_\_global\_\_ void kernel\_DrugSimulation(double \*d\_ic50, double \*d\_cvar, double \*d\_conc, double \*d\_CONSTANTS, double \*d\_STATES, double \*d\_RATES, double \*d\_ALGEBRAIC,

double \*d\_STATES\_RESULT,

// double \*time, double \*states, double \*out\_dt, double \*cai\_result,

// double \*ina, double \*inal,

// double \*ical, double \*ito,

// double \*ikr, double \*iks,

// double \*ik1,

unsigned int sample\_size,

cipa\_t \*temp\_result, cipa\_t \*cipa\_result,

param\_t \*p\_param

);

\_\_device\_\_ void kernel\_DoDrugSim(double \*d\_ic50, double \*d\_cvar, double d\_conc, double \*d\_CONSTANTS, double \*d\_STATES, double \*d\_RATES, double \*d\_ALGEBRAIC,

double \*d\_STATES\_RESULT,

// double \*time, double \*states, double \*out\_dt, double \*cai\_result,

// double \*ina, double \*inal,

// double \*ical, double \*ito,

// double \*ikr, double \*iks,

// double \*ik1,

double \*tcurr, double \*dt, unsigned short sample\_id, unsigned int sample\_size,

cipa\_t \*temp\_result, cipa\_t \*cipa\_result,

param\_t \*p\_param

);

#endif

In the post-processing mode (the mode that only runs the simulation once and take all biomarkers calculation), these comments were uncommented to let more data flow in and out of the GPU kernel.

## param.cpp and param.hpp

These scripts manages the simulation parameters. It reads the input deck file and put them into the code as variables. The param.cpp implements functions to read the input file, load, parse, validate parameter files, and act as a failsafe if one or more parameters or parameter file is not readable. This script has the default values and path of each required numbers and files. This file ensures that the simulation runs with accurate and user-defined settings. Header of param.hpp declares variables used to store these parameter values and ensuring that they are well defined in the simulation. The header file goes as a following struct:

#ifndef PARAM\_HPP

#define PARAM\_HPP

struct param\_t

{

unsigned short simulation\_mode; // toggle between sample-based or full-pace simulations

bool is\_dutta; // TRUE if using Dutta scaling

unsigned short gpu\_index;

bool is\_print\_graph; // TRUE if we want to print graph

bool is\_using\_output; // TRUE if using last output file

bool is\_cvar;

double bcl; // basic cycle length

// unsigned int max\_samples;

unsigned short pace\_max; // maximum pace

unsigned short find\_steepest\_start;

unsigned short celltype; // cell types

double dt; // time step

double dt\_write; // writing step

double inet\_vm\_threshold; // Vm threshold for calculating inet

char hill\_file[1024];

char cvar\_file[1024];

char drug\_name[100];

char concs[100];

float conc;

void init();

void show\_val();

};

#endif

And the main file reads parameter and their default values to the whole project with following way:

#include "param.hpp"

#include <cstdio>

#include "glob\_funct.hpp"

void param\_t::init()

{

simulation\_mode = 0;

// max\_samples = 10000;

is\_dutta = true;

gpu\_index = 0;

is\_cvar = false;

bcl = 2000.;

pace\_max = 10;

find\_steepest\_start = 5;

celltype = 0.;

dt = 0.005;

// dt = 0.1;

conc = 99.0;

// dt\_write = 2.0;

// inet\_vm\_threshold = -88.0;

snprintf(hill\_file, sizeof(hill\_file), "%s", "./drugs/bepridil/IC50\_samples.csv");

snprintf(cvar\_file, sizeof(cvar\_file), "%s", "./drugs/10000\_pop.csv");

snprintf(drug\_name, sizeof(drug\_name), "%s", "bepridil");

// snprintf(concs, sizeof(concs), "%s", "99.0");

}

void param\_t::show\_val()

{

//change this to printf somehow

mpi\_printf( 0, "%s -- %s\n", "Simulation mode", simulation\_mode ? "full-pace" : "sample-based" );

mpi\_printf( 0, "%s -- %s\n", "Hill File", hill\_file );

mpi\_printf( 0, "%s -- %hu\n", "Celltype", celltype);

mpi\_printf( 0, "%s -- %s\n", "Is\_Dutta", is\_dutta ? "true" : "false" );

mpi\_printf( 0, "%s -- %s\n", "Is\_Cvar", is\_cvar ? "true" : "false" );

mpi\_printf( 0, "%s -- %lf\n", "Basic\_Cycle\_Length", bcl);

mpi\_printf( 0, "%s -- %d\n", "GPU\_Index", gpu\_index);

mpi\_printf( 0, "%s -- %hu\n", "Number\_of\_Pacing", pace\_max);

mpi\_printf( 0, "%s -- %hu\n", "Pace\_Find\_Steepest", find\_steepest\_start);

mpi\_printf( 0, "%s -- %lf\n", "Time\_Step", dt);

mpi\_printf( 0, "%s -- %s\n", "Drug\_Name", drug\_name);

mpi\_printf( 0, "%s -- %lf\n\n\n", "Concentrations", conc);

}

## Critical Scripts

This section highlights the key scripts within the framework, focusing on their roles and significance in executing and customising simulations. These critical scripts form the backbone of the codebase, enabling efficient management of computations, GPU operations, and parameter configurations. If there is any issue with the code, 80% of the issue can be solved by looking at these three script first. Most of debugging, adjustement, and modification happen in these scripts. Below is a detailed breakdown:

## Main script

The main script (main.cu) serves as the central controller of the simulation framework. It is the entry point of the program, responsible for managing the simulation's overall workflow, from initialisation to the final output. Below are its primary responsibilities and we can lookup into the main script if we need some modification related to:

* Command-Line Argument Parsing
  + Reads user-provided inputs, such as input\_deck.txt, initial state files, and IC50/Hill coefficient files.
  + Validates the correctness and presence of required input files.
  + Flags any missing or improperly formatted files, preventing the simulation from starting.
* Simulation Setup
  + Loads and processes input parameters from the provided files.
  + Sets up the pacing protocol, drug concentrations, and simulation-specific configurations.
  + Handles initialisation for multiple samples and conditions.
* Memory Management
  + Allocates and deallocates memory for the CPU and GPU, ensuring efficient resource utilisation using core (thread) per block calculation.
  + Transfers data such as parameters, initial states, and configurations between host (CPU) and device (GPU).
* GPU Kernel Launch
  + Initiates GPU computations by invoking the kernel functions for parallel execution.
  + Ensures proper thread and block configurations to optimise performance for the given workload.
* Result Handling
  + Collects output data from the GPU and processes it for storage.
  + Saves simulation results to the designated output directory.
  + Generates logs for debugging and validation purposes.
* Error Handling and Debugging
  + Includes checks to identify and report common errors, such as missing inputs, memory allocation failures, or kernel launch issues. If something is wrong with the kernel function in the GPU, the code always goes back to this script, assuming the kernel function has done even with no output.
  + Provides detailed logs to help trace and resolve errors during simulation execution.

## GPU control script

The GPU control script (gpu.cu and gpu.cuh) is the core component that drives the high-performance computations. It includes all GPU-specific operations, such as kernel implementations, memory allocation, and parallel execution logic. This script manages the distribution of workloads across GPU threads and blocks, ensuring that simulations are processed efficiently.

Key responsibilities of this script include:

* Implementing the numerical solvers (e.g., Forward Euler, Rush-Larsen) for simulating cardiac cell dynamics.
* Handling memory transfers between the host (CPU) and device (GPU), including input parameters, initial states, and output data.
* Optimising performance through techniques like shared memory utilisation and load balancing.

Most performance issues or GPU errors, such as kernel launch failures or memory overflows, can be traced back to this script. It is also the primary location for implementing modifications to the numerical methods or adding new solvers.

## Parameters

The parameter script (param.cpp and param.hpp) defines and manages the input parameters required for the simulation. It acts as a configuration layer, ensuring that all essential variables are correctly initialised and passed to the simulation workflow. This script is also responsible for providing default values for parameters when they are not explicitly defined in the input files. It ensures that the simulation remains robust against incomplete or inconsistent configurations.

Adjustments to this script are necessary when:

* Adding new features or variables to the simulation.
* Customising the simulation for different biological models or experimental setups.
* Debugging errors related to undefined or mismatched parameters.

## Commands and Flag Usages

This section outlines the essential commands and flags used to execute, customise, and manage the simulation framework. This part tend to face human error as in wrong path. The error usually shown as the file being read as 0, or file not found. Also the code being run without proper compilation previously. The first phase of the simulation’s command and flags run as below:

./drug\_sim -input\_deck input\_deck.txt -hill\_file drug/IC50\_file.csv

Second phase of the simulation can be run as below:

./drug\_sim -input\_deck input\_deck.txt -hill\_file drug/IC50\_file.csv -init\_file initfile.csv

The commands above represent two primary phases of the simulation process. In the first phase, the simulation is initiated with the essential input deck and Hill coefficient file. The -input\_deck flag specifies the configuration file (input\_deck.txt), which contains simulation parameters such as pacing details, cell models to be used, and output settings. The -hill\_file flag points to a CSV file (drug/IC50\_file.csv) containing IC50 and Hill coefficient values for the drug being simulated. This step generates initial conditions and baseline data required for further simulations.

In the second phase, the simulation proceeds with additional parameters specified by the -init\_file flag. This flag allows the user to include an initial state file (initfile.csv), which represents the system's state at the end of the first phase. By leveraging this initial state, the second phase can bypass reinitialisation and focus on computing drug effects or specific pacing protocols. This phase is particularly useful for testing drug interactions or extended pacing simulations without repeating initial computations.

In other cell models, more input is required and uses the same method to be registered in the simulation. ORd 2017 requires additional -herg\_file as input, since this newer cell model needs herg fitting value from CiPA. This update increases the simulation accuracy. A common ORd 2017 simulation will be called as such:

./drug\_sim -input\_deck input\_deck.txt -hill\_file drug/IC50\_file.csv -herg\_file drug/herg.csv

While herg.csv is structured as:

Kmax,Ku,n,halfmax,Vhalf,slope

5594000,0.0001719,0.9374,147200000,-61.34,NA

Each drug will have their own hERG file.

Also in three of the cell models, future update related to inter-individual variability will be applied. In the future, these cell models will require additional -cvar\_file (conductance variability) as input. This update will enrich the simulation capability to simulate drug effect in different population segments such as healthy people compared with people living with heart failure history. This update will increase the simulation accuracy and variability. A common ORd 2017 simulation with inter-individual variability will be called as such:

./drug\_sim -input\_deck input\_deck.txt -hill\_file drug/IC50\_file.csv -herg\_file drug/herg.csv -cvar\_file cvar.csv

## Troubleshooting

This section provides guidance on addressing frequent issues encountered when using the simulation framework. These troubleshooting steps are designed to assist users in diagnosing and resolving problems effectively. Especially in the situation where CUDA debugger were not informative enough due to the uniqueness of the parallelisation.

* File Not Found Errors

Problem: The simulation fails with an error indicating that a required file is missing or cannot be accessed.

Cause:

* + Incorrect file path provided in the command-line arguments.
  + The file does not exist in the specified directory.

Solution:

Verify the file path in the command. Ensure the paths match the directory structure (e.g., drug/IC50\_file.csv). Check if the file exists in the specified folder. If not, create or move the file to the correct location. Ensure file permissions allow read access.

* Compilation Errors

Problem: Compilation fails, showing errors related to missing libraries or incompatible flags.

Cause:

* + Required CUDA or GCC/G++ versions are not installed.
  + Incorrect flags in the Makefile.

Solution:

Verify that the system has the correct version of CUDA (e.g., version 11.x or later).Check that nvcc and gcc are installed and added to the system's PATH. Review and update the Makefile to match the system configuration. For instance, ensure paths to CUDA libraries and headers are correct.

* Zeroes or Incomplete Output

Problem: The output files contain no data or are incomplete.

Cause:

* + Incorrect input parameters or missing initial state files.
  + Errors in the numerical solver or GPU execution.

Solution:

Ensure the input deck (input\_deck.txt) is correctly formatted and contains all required parameters.Verify that the initial state file (initfile.csv) exists and is properly formatted. Check the GPU kernel execution logs (printed in the terminal) for errors (e.g., memory overflows or segmentation faults).

* GPU Kernel Launch Failures,

Problem: The simulation crashes during execution, or right just before interacting with GPU kernel function.

Cause:

* + Insufficient GPU memory to handle the workload.
  + Incompatible GPU architecture flags in the Makefile.

Solution:

Reduce the number of samples or concentrations in the input parameters to fit within available GPU memory. For information, in average it takes 2 MB of GPU memory to simulate one sample. Verify the GPU's compute capability and ensure the Makefile includes the correct -arch flag (e.g., -arch=sm\_86 for NVIDIA RTX 4090).

* Runtime Errors Due to Command-Line Arguments

Problem: The program crashes or produces unexpected results when running with specific arguments.

Cause:

* + Missing or incorrectly formatted command-line arguments.

Solution:

Ensure all required arguments (-input\_deck, -hill\_file, -init\_file, etc.) are included and point to valid files. Refer to the section on "Commands and Flag Usages" for examples of valid commands.

* Mismatched Results

Problem: The GPU simulation results do not match the expected outputs from the CPU simulation.

Cause:

* + Numerical instability or precision issues in the GPU solver.
  + Issue with drug effect implementation

Solution:

Double-check the numerical solver configurations to ensure consistency between CPU and GPU implementations. Compare the input data files to confirm they are identical for both simulations. Start comparison from non-drug, control data first. If control from both CPU and GPU shows no noticeable difference in action potential shape, it is confirmed that the issue comes from drug effect implementation.

Check the IC50 input, current development applies these values to the ratio of the gate instead of the gate. Previous implementation may implement directly to the gate. We can differentiate these two IC50 files by looking at the orde of the numbers. Old implementation method usually uses thousands or hundred of thousands, while more recent version usually stays at most around the thousands.

If the IC50 uses a more recent version, ensure in the drug effect implementation function, it does not change the gate, but instead the \_b of the gate (GNaL\_b for example).

* Slow Performance on GPU

Problem: The GPU simulation runs slower than expected, or it runs more than 24 even 48 hours for 1000 pacing or less.

Cause:

* + Inappropriate workload size or insufficient parallelisation.
  + Suboptimal GPU memory management.
  + Less compatible GPU clock speed.

Solution:

Increase the workload size to fully utilise GPU cores (e.g., simulate more samples or concentrations). Profile the simulation using tools like nvprof or Nsight to identify bottlenecks in memory transfers or kernel executions. Ensure the simulation runs on a more recent GPU generation from NVIDIA, specifically the RTX series (RTX 3080Ti, RTX 4090, etc.). Also for information, this simulation tested and optimised for a gaming grade PC GPU with clock speed around 900-1200 MHz. Some of server-grade GPU has a lot of GPU memory but lacking of clockspeed. We conducted a test on a server grade GPU, NVIDIA T4 with 81 GB of GPU memory, but it was considered too slow. It finished the job simulating 100 pacing of ORd 2011 with drug effect in around 36-40 hours.

* Memory Overflows or Leaks

Problem: The simulation crashes with strange, unreadable error message, makes us cannot access the GPU, or hangs due to excessive memory usage.

Cause:

* + Improper memory allocation or deallocation in GPU kernels.

Solution:

Inspect the GPU control script (gpu.cu) for memory management issues.

Use tools like cuda-memcheck to debug memory leaks or access violations.

* Compatibility Issues

Problem: The code does not work on certain hardware or software configurations. Signed by a fail in running the kernel script with no error at all.

Cause:

* + Dependency mismatches, such as outdated libraries or drivers.

Solution:

Update the GPU drivers to the latest version recommended by NVIDIA. Ensure the CUDA version matches the code’s requirements (CUDA 11.x). Also ensure the Makefile includes the correct -arch flag (e.g., -arch=sm\_86 for NVIDIA RTX 4090). The code is not tested on other than GTX or RTX family. The oldest GPU we've ever tested the code was GTX 1660 Ti.

|  |
| --- |
| Enhancing the efficiency of animal-alternative *in-silico* drug cardiotoxicity prediction through CUDA-based parallel processing |
|  |
| Thesis for Master of Engineering |
| December 2024 |
| Iga Narendra Pramawijaya |