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

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

|  |
| --- |
| Approval of the Thesis for the Master of Engineering Submitted by Ali Ikhsanul Qauli |
|  |
| December 2020 |
|  |
|  |
|  |
|  |
| C:\Users\Med IT\Pictures\Other\thesis\sign2.pngChair of Committee Kwang Soup Song (seal) |
|  |
| C:\Users\Med IT\Pictures\Other\thesis\sign5.pngCommittee Ki Moo Lim (seal) |
|  |
| Committee Hojong Choi (seal) |
|  |
| Graduate School  Kumoh National Institute of Technology |
|  |

|  |
| --- |
|  |
| 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: The comprehensive *in vitro* proarrhythmia assay (CiPA) has opened broad opportunities to incorporate *in silico* experiment as an integral part of drug assessment. Following that, the initial assessment of the Hill coefficient and IC50, which are two important parameters describing drug cellular dynamics, can be included in the in silico experiment as well. The initial assessment consists of Hill fitting and parameters bootstrapping. In this work, I propose a new approach to deploy simple Monte Carlo (MC) simulation with Gaussian scatter incorporated into the calculation of bootstrap samples.

Method: The experimental dose-response data is processed into Hill fitting with the least square method (LSM) to obtain best-fit parameters. These fitted parameters combined with Gaussian scatter of blocking data will act as a data generator for MC simulation that will result in bootstrap samples of Hill coefficient and IC50. Finally, several aspects of those bootstrap samples are assessed such as the 95% confidence interval, the parameter distribution (in the form of the histogram), and finally sensitivity analysis by predicting blocking response from various drug concentrations.

Results: It is found that for insufficient dose-response data, both the proposed MC algorithm and the existing Markov chain Monte Carlo (MCMC) method can jump to the quite high value of uncertainties faster especially with observed mean blocking percentage below 60%. However, both algorithms can yield a similar maximum uncertainty profile except within a region of 15-25% minimum observed mean blocking where the MC method generated higher uncertainty than MCMC. In addition, I was able to show that for sufficient observed experimental data where allowed drug dosage is within this range, the proposed MC method match quite well with existing MCMC in term of distribution of fitted parameters.

Conclusion: The proposed MC method for bootstrapping Hill coefficient and IC50 is quite simpler to implement than the existing MCMC method as it adds only Gaussian sampling into the main Hill fitting algorithm. In addition to its simplicity, the proposed MC method has comparable results with the existing MCMC method in terms of fitted parameter distribution and uncertainty quantification.

|  |
| --- |
|  |

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

소개 : 종합 체외 부정맥 분석(CiPA)은 약물 평가의 필수 부분으로 In silico 컴퓨터 시뮬레이션을 통합할 수 있는 광범위한 기회를 열었다. 그에 따라 약물에 의한 세포 역학을 설명하는 두 가지 중요한 매개 변수인 Hill계수와 IC50의 초기 평가는 In silico 실험에 포함될 수 있다. 이 초기 평가는 Hill 피팅과 매개 변수의 부트스트랩과정으로 구성된다. 본 연구에서는 부트스트랩 샘플의 계산을 위해 가우스 산포가 포함된 몬테카를로(MC) 시뮬레이션을 사용하는 새로운 접근방식을 제안하였다.

방법 : 최소제곱법(LSM)을 사용한 힐 피팅을 통해 실험 복용량-반응 데이터로부터 최적의 매개변수를 생성하였다. 약물에 의한 차단 데이터의 가우스 산포와 결합된 피팅 된 매개변수는 MC 시뮬레이션을 위한 데이터 생성기로 사용되어 Hill계수와 IC50의 부트스트랩 샘플을 생성한다. 부트스트랩 된 샘플들은 95%의 신뢰구간에 분포하는 정도, 매개 변수의 분포(히스토그램의 형태) 및 다양한 약물 농도에 따른 차단 반응의 예측을 통한 민감도 분석을 통해 평가되었다.

결과 : 복용량-반응 데이터가 충분하지 않은 경우 MC 알고리즘과 Markov Chain Monte Carlo (MCMC) 알고리즘 모두에서 매우 높은 불확실성 수치를 생성하였으며, 이는 특히 측정된 평균 차단 비율이 60% 미만인 경우에서 두드러지게 관찰되었다. 하지만, 두 알고리즘 모두 유사한 최대 불확실성 프로파일을 생성할 수 있었으며 최소 측정된 평균 차단 정도가 15~25%인 영역에서는 MC 방법이 MCMC 방법보다 더 높은 불확실성을 생성하였다. 또한, 허용된 약물 복용량이 이 범위 내에서 측정된 실험 데이터가 충분한 경우 연구에서 제안한 MC 방법이 피팅 된 매개 변수의 분포 측면에서 기존의 MCMC 방법과 매우 잘 일치 하는 것을 보여주었다.

결론 : Hill 계수와 IC50을 부트스트랩하기 위해 연구에서 제안한 MC 방법은 주 Hill 피팅 알고리즘에 가우시안 샘플링 과정이 추가되기 때문에 기존의 MCMC 방법에 비해 구현이 쉽다. 이러한 단순성 외에도 제안된 MC방법은 적합 매개 변수의 분포 및 불확실성의 정량화 측면에서 MCMC 방법과 유사한 결과를 낼 수 있었다.

**Contents**

[[List of Figures] iii](#_Toc183181779)

[[List of Tables] iii](#_Toc183181780)

[[Glossary] iv](#_Toc183181781)

[[Acknowledgement] v](#_Toc183181782)

[Chapter 1. Introduction 1](#_Toc183181783)

[1.1 In-Silico Electrophysiology Simulation 1](#_Toc183181784)

[1.2 Parallel Computing 2](#_Toc183181785)

[1.2.1 Central Processing Unit (CPU) for Parallel Computing 3](#_Toc183181786)

[1.2.2 Graphics Processing Unit (GPU) for Parallel Computing 4](#_Toc183181787)

[1.2.3 CUDA 4](#_Toc183181788)

[1.2.4 CellML 5](#_Toc183181789)

[1.3 Previous Study 7](#_Toc183181790)

[1.4 Objectives 8](#_Toc183181791)

[Chapter 2. Methodologies 8](#_Toc183181792)

[2.1 Generating C Code from CellML 8](#_Toc183181793)

[2.2 GPU Memory Adjustment and Offseting 10](#_Toc183181794)

[2.3 Solving Ordinary Differential Equations 12](#_Toc183181795)

[2.4 Simulation Protocol and Code Organisation 14](#_Toc183181796)

[2.5 Output Format 17](#_Toc183181797)

[2.6 Compilation, Input Files, and Testing 18](#_Toc183181798)

[2.6.1 Compiling with makefile 18](#_Toc183181799)

[2.6.2 Required Input Files 20](#_Toc183181800)

[2.6.3 Testing and Result Validation Method 21](#_Toc183181801)

[Chapter 3. Results and Discussion 23](#_Toc183181802)

[3.1 GPU Simulation Result Using ORd 2011 Cell Model 23](#_Toc183181803)

[3.1.1 Result Validation 24](#_Toc183181804)

[3.1.2 Result Validation Under Drug 24](#_Toc183181805)

[3.1.3 Computational Time and Efficiency Analysis 24](#_Toc183181806)

[3.2 GPU Simulation Result Using ORd 2017 Cell Model 25](#_Toc183181807)

[3.2.1 Result Validation 25](#_Toc183181808)

[3.2.2 Result Validation Under Drug 25](#_Toc183181809)

[3.2.3 Computational Time and Efficiency Analysis 25](#_Toc183181810)

[3.3 GPU Simulation Result Using Tomek Cell Model 25](#_Toc183181811)

[3.3.1 Result Validation 25](#_Toc183181812)

[3.3.2 Result Validation Under Drug 26](#_Toc183181813)

[3.3.3 Computational Time and Efficiency Analysis 26](#_Toc183181814)

[Chapter 4. Conclusion and Limitation 27](#_Toc183181815)

[[References] 28](#_Toc183181816)

[Appendix 30](#_Toc183181817)

[A. Code for Data processing 30](#_Toc183181818)

[B. Code for linear and nonlinear LSM 32](#_Toc183181819)

[C. Code for Monte Carlo simulation 35](#_Toc183181820)

[D. Complete code for Hill fitting with LSM and Monte Carlo simulation 36](#_Toc183181821)

# [List of Figures]

**No table of figures entries found.**

**No table of figures entries found.**

[[Figure 2.1] OpenCOR interface when selecting Tomek model and converting it to C codes on MacOS. 9](#_Toc183182480)

[[Figure 2.2] Main difference in values storing paradigm after CUDA-parallelisation, assuming column size is 13. 11](#_Toc183182481)

[[Figure 3.1] Action Potential Shape of both CPU (dashed orange) and GPU (blue) Simulation Result 24](#_Toc183182482)

# [List of Tables]

[[Table 2. 1] 12 CiPA training drugs and its TdP risk classification **Error! Bookmark not defined.**](#_Toc57331937)

[[Table 3. 1] Hill equation parameters and confidence intervals (square bracket) for the 12 CiPA training drugs with scheme 1 approach from [Figure 2. 1] **Error! Bookmark not defined.**](#_Toc57331935)

[[Table 3. 2] Hill equation parameters and confidence intervals (square bracket) for the 12 CiPA training drugs with scheme 2 approach from [Figure 2. 1] **Error! Bookmark not defined.**](#_Toc57331936)

# [Glossary]

|  |  |
| --- | --- |
| TdP | Torsade de pointes |
| CiPA | Comprehensive *in vitro* Proarrhythmia Assay |
| ECG | Electrocardiogram |
| UQ | Uncertainty quantification |
| MC | Monte Carlo |
| MCMC | Markov chain Monte Carlo |
| DRAM | Delaying rejection adaptive Metropolis |
| AARJ | Adaptive automatic reversible jump |
| LSM | Least square method |
| LM | Lavenberg-Marquardt |
| GSL | GNU Scientific library |
| N/A | Not applicable |
| 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 |
|  |  |

# [Acknowledgement]

In the name of God, The Most Gracious and The Most Merciful. Alhamdulillah, all praises to God. This thesis would not have been completed without God’s will and guidance. Also, may blessing and peace always be upon Prophet Muhammad, his family, his companion, and his followers all over the world.

First of all, I would like to express my sincere gratitude to my supervisor Professor Ki Moo Lim. Thank you for allowing me to be part of the research member of the Computational Medicine Lab (CML). Your valuable guides and support really help me excel in many aspects I am good at during my study and research in CML. Also, I would like to thank the referee Professor Kwang Soup Song, and Professor Ho Jong Choi for their time, support, and constructive comments.

I would also like to extend my gratitude to all lab mates that I have ever met and collaborated with; Aulia, Getu, Abebe, 다운,하영, Marcell,준효, and 에담. Thank you for all support and advice that had been given to me. I do not think I can survive during my study and research without your help, guys.

For all Indonesian students that I have ever met and collaborated with during my study in Korea, especially those who stayed in Shinpyeong; Aulia, Rahma, Salwa, Naufal, other students from KIT, also from Indonesian Students Association in South Korea (Perpika); Anafi, CQ, and other students from various campuses and cities in Korea that I cannot include here one by one, thank you to be important parts of my life. You really help me excel to become a better human being as a whole. Take care and stay healthy, guys.

Lastly, I wish to express my gratitude to my family, my dad, mom, brother, and sisters. They have always unconditionally supported me to be as best as I can be. Thank you for all that support. May The Almighty God grants you the best life in this small, temporary world and in the eternal afterlife.

# 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 minimize 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 programming model designed for parallel computing on shared-memory architectures, typically used to exploit multicore CPUs. It enables developers to add parallelism to existing C, C++, and Fortran code using simple compiler directives (pragmas). OpenMP divides tasks across multiple threads that share the same memory space, allowing for straightforward parallelization of loops and sections of code with minimal modifications. OpenMP is relatively easy to implement, making it an excellent choice for applications requiring high performance on a single multicore processor, where shared memory among threads simplifies data access.

MPI (Message Passing Interface), on the other hand, is a standard used for parallel computing on distributed-memory systems, such as clusters or supercomputers. MPI allows multiple processes running on separate memory spaces to communicate by sending and receiving messages, making it suitable for applications where tasks need to run on different nodes in a network. This model is more complex than OpenMP, as it requires explicit data sharing, but it offers greater flexibility and scalability, allowing parallelization across a large number of processors. MPI is ideal for high-performance computing tasks that involve significant data exchange across nodes, making it the go-to solution for 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. By combining these technologies, developers can create highly efficient applications that fully exploit the power of modern multi-core and distributed computing architectures. This enables significant performance gains in various scientific, engineering, and industrial applications.

### 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. 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 APIs and 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.

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. 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. 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. [[cite](https://doi.org/10.1098%2Frsta.2008.0094)] [[cite](https://doi.org/10.1093%2Fbioinformatics%2Fbtn390)] [[cite](http://scholar.google.com/scholar_lookup?&title=BioModels%20Database%3A%20a%20free%2C%20centralized%20database%20of%20curated%2C%20published%2C%20quantitative%20kinetic%20models%20of%20biochemical%20and%20cellular%20systems&publication_year=2006&author=Le%20Novere%2CN&author=Bornstein%2CB&author=Broicher%2CA&author=Courtot%2CM&author=Donizelli%2CM&author=Dharuri%2CH&author=Li%2CL&author=Sauro%2CH&author=Schilstra%2CM&author=Shapiro%2CB)]

## Previous Study

Parallelisation in computational biology is not an entirely new concept. The Cells in Silico (CiS) framework presented by Berghoff et al. (2020) [[cite](https://link.springer.com/article/10.1186/s12859-020-03728-7)] 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 Miguel, et al [[cite](https://www.sciencedirect.com/science/article/abs/pii/S0167739X19308817)] 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 a 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 [[cite](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4425459/)]. As a result, McIntosh-Smith et, al. achieved of 46% at peak, or 1.43 TFLOP/s on a single Nvidia GTX 680.

Barth 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 [[cite](https://www.researchgate.net/publication/281886386_Parallel_Biological_In_Silico_Simulation)].

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. (2019) 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​ [2]. Additionally, the integration of multi-scale models, from cellular-level simulations to 3D anatomical representations, has expanded the applicability of these tools, enabling more comprehensive analyses of cardiac arrhythmogenesis and therapy optimization​ [2]. [⁠In silico Assessment ofPharmacotherapy for Human Atrial Patho-Electrophysiology Associated With hERG-Linked Short QT Syndrome - Dominic G. Whittaker, et all]

## Objectives

The objective of this study is to enhance the efficiency and scalability of *in-silico* simulations for predicting cardiovascular drug toxicity 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, 2017 and Tomek. 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 Tomek 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.

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. Initially, we aimed to generalize the use of the Rush-Larsen method across all cell models. However, when applied to ORd 2017 and Tomek 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 Tomek models, it proved to be computationally intensive, significantly increasing the runtime.

To optimize the parallelization 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 Tomek 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 Tomek 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 Tomek, 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 Tomek 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 number, 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, and ca\_dia. 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. Tomek 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, fall within an acceptable range of [...]. 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 Tomek. 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 Tomek 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, 66, 99, and 132 mMol. 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. Validation also involved statistical measure such as mean absolute error (MAE) to quantify differences between the GPU-based and OpenCOR results.

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 3.3] Action Potential Shape of both CPU (blue) and GPU (dashed 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

### Computational Time and Efficiency Analysis

## 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. Key biomarkers, such as action potential duration, ionic currents, and calcium transients, 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 3.2 provides a visual comparison of the action potentials produced by the GPU and CPU simulations, illustrating their near-identical behaviour.



[Figure 3.3] Action Potential Shape of both CPU (dashed orange) and GPU (blue) Result Using ORd 2017

### 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. Despite the added complexity of drug effects, the GPU simulation produced outputs that were nearly identical to those of the CPU-based OpenCOR simulations.

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.

[Figure 3.3] Action Potential Shape of both CPU (dashed orange) and GPU (blue) Result Using ORd 2011

### Computational Time and Efficiency Analysis

## GPU Simulation Result Using Tomek Cell Model

This section highlights the results obtained from GPU simulations using the Tomek 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 Tomek model, it results in longer computational times compared to the more efficient Rush-Larsen method used in the ORd 2011 simulations. Despite this limitation, the forward Euler method ensures that the GPU simulation remains stable and accurate for this model.

### Result Validation

The validation process for the Tomek 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.

The results 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 Tomek cell model, even when employing the forward Euler method. Figure 3.3 illustrates the action potential traces from both GPU and CPU simulations, demonstrating their near-identical nature.

[Figure 3.3] Action Potential Shape of both CPU (dashed orange) and GPU (blue) Result Using ORd 2011

### Result Validation Under Drug

The validation of the GPU-based simulation for the Tomek 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.

The results 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 Tomek 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.

[Figure 3.3] Action Potential Shape of both CPU (dashed orange) and GPU (blue) Result Using ORd 2011

### Computational Time and Efficiency Analysis

# 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 50 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, making them better representations of human physiology. Testing the methodology on diverse GPU hardware models would provide insights into performance variations across systems, ensuring broader applicability. Integrating artificial intelligence techniques could further refine result analysis and optimise simulation parameters, enhancing both accuracy and efficiency. The method’s potential for industrial application is also notable, particularly in pharmaceutical pipelines where it could support large-scale drug development.

Real-world testing in laboratory environments is essential to validate the simulation's reliability under varied conditions. 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. Hill, A.V., *The possible effects of the aggregation of the molecules of hæmoglobin on its dissociation curves.* The Journal of Physiology, 1910. **40**: p. i--vii.

2. Gesztelyi, R., et al., *The Hill equation and the origin of quantitative pharmacology.* Archive for History of Exact Sciences, 2012. **66**(4): p. 427-438.

3. Langmuir, I., *THE ADSORPTION OF GASES ON PLANE SURFACES OF GLASS, MICA AND PLATINUM.* Journal of the American Chemical Society, 1918. **40**(9): p. 1361-1403.

4. Neubig, R.R., et al., *International Union of Pharmacology Committee on Receptor Nomenclature and Drug Classification. XXXVIII. Update on terms and symbols in quantitative pharmacology.* Pharmacol Rev, 2003. **55**(4): p. 597-606.

5. Kenakin, T., *Data-driven analysis in drug discovery.* J Recept Signal Transduct Res, 2006. **26**(4): p. 299-327.

6. Motulsky, H. and A. Christopoulos. *Fitting Models to Biological Data Using Linear and Nonlinear Regression: A Practical Guide to Curve Fitting*. 2004.

7. Black, J.W. and P. Leff, *Operational Models of Pharmacological Agonism.* Proceedings of the Royal Society of London. Series B, Biological Sciences, 1983. **220**(1219): p. 141-162.

8. Keller, F., et al., *PK-PD curve-fitting problems with the Hill equation? Try one of the 1-exp functions derived from Hodgkin, Douglas or Gompertz.* Int J Clin Pharmacol Ther, 2002. **40**(1): p. 23-9.

9. Giraldo, J., et al., *Assessing the (a)symmetry of concentration-effect curves: empirical versus mechanistic models.* Pharmacol Ther, 2002. **95**(1): p. 21-45.

10. Van der Graaf, P.H. and W.B. Stam, *Analysis of receptor inactivation experiments with the operational model of agonism yields correlated estimates of agonist affinity and efficacy.* Journal of Pharmacological and Toxicological Methods, 1999. **41**(2): p. 117-125.

11. Gintant, G.A., *Preclinical Torsades-de-Pointes screens: advantages and limitations of surrogate and direct approaches in evaluating proarrhythmic risk.* Pharmacol Ther, 2008. **119**(2): p. 199-209.

12. Sager, P.T., et al., *Rechanneling the cardiac proarrhythmia safety paradigm: a meeting report from the Cardiac Safety Research Consortium.* Am Heart J, 2014. **167**(3): p. 292-300.

13. Vicente, J., et al., *Mechanistic Model-Informed Proarrhythmic Risk Assessment of Drugs: Review of the "CiPA" Initiative and Design of a Prospective Clinical Validation Study.* Clin Pharmacol Ther, 2018. **103**(1): p. 54-66.

14. Strauss, D.G., et al., *Comprehensive In Vitro Proarrhythmia Assay (CiPA) Update from a Cardiac Safety Research Consortium / Health and Environmental Sciences Institute / FDA Meeting.* Ther Innov Regul Sci, 2019. **53**(4): p. 519-525.

15. Chang, K.C., et al., *Uncertainty Quantification Reveals the Importance of Data Variability and Experimental Design Considerations for in Silico Proarrhythmia Risk Assessment.* Front Physiol, 2017. **8**: p. 917.

16. Haario, H., et al., *DRAM: Efficient adaptive MCMC.* Statistics and Computing, 2006. **16**(4): p. 339-354.

17. Laine, M. and J. Tamminen, *Aerosol model selection and uncertainty modelling by adaptive MCMC technique.* Atmos. Chem. Phys., 2008. **8**(24): p. 7697-7707.

18. Soetaert, K. and T. Petzoldt, *Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME.* Journal of statistical software, 2010. **33**: p. 1.

19. Crumb, W.J., Jr., et al., *An evaluation of 30 clinical drugs against the comprehensive in vitro proarrhythmia assay (CiPA) proposed ion channel panel.* J Pharmacol Toxicol Methods, 2016. **81**: p. 251-62.

20. Levenberg, K., *A METHOD FOR THE SOLUTION OF CERTAIN NON-LINEAR PROBLEMS IN LEAST SQUARES.* Quarterly of Applied Mathematics, 1944. **2**(2): p. 164-168.

21. Marquardt, D.W., *An Algorithm for Least-Squares Estimation of Nonlinear Parameters.* Journal of the Society for Industrial and Applied Mathematics, 1963. **11**(2): p. 431-441.

22. Li, Z., et al., *Improving the In Silico Assessment of Proarrhythmia Risk by Combining hERG (Human Ether-a-go-go-Related Gene) Channel-Drug Binding Kinetics and Multichannel Pharmacology.* Circ Arrhythm Electrophysiol, 2017. **10**(2): p. e004628.

# Appendix

## Code for Data processing

Prior to fitting procedure, there will be data processing steps that read experimental data, store it into arrays, sort it with respect to channel name, and average blocking data on each channel. These steps are described by several C++ functions and steps as follow:

* Read experimental data and find amount of data per channel

int readDataPerChannel(int numberOfData, std::string drugName, std::string channelName, std::string\*\* dataStr){

int iter;

iter = 0;

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

if(dataStr[i][0]==drugName && dataStr[i][2]==channelName){

iter = iter + 1;

}

}

return iter;

}

This function will return an integer value of the number of data on each channel of the drug.

* Store the data per channel into array

void storeDataPerChannel(int numberOfData, std::string drugName, std::string channelName, std::string\*\* dataStr,double \*\*dataPerChannel, double \*\*data){

int iter;

iter = 0;

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

if(dataStr[i][0]==drugName && dataStr[i][2]==channelName){//change channelList index for other channel

dataPerChannel[iter][0] = data[i][0];//conc

dataPerChannel[iter][1] = data[i][1];//block

iter = iter + 1;

}

}

return;

}

The output of this function is the dataPerChannel array that contains all experimental data per channel.

* Reading data per channel to obtain number of data having same amount of drug concentration

int readConcPerChannel(int numberOfDrugDataPerChannel, double \*\*dataPerChannel){

int iter;

double conc;

iter = 0;

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

if(i==0){

conc = dataPerChannel[i][0];//conc variable

iter = iter + 1;

}

else{

if(dataPerChannel[i][0]!=conc){

conc = dataPerChannel[i][0];//conc variable

iter = iter + 1;

}

}

}

return iter;

}

It will return an integer value of the number of different concentrations within the data per channel.

* Calculating the averaged data for each drug concentration as described in **Error! Reference source not found.**]

void findAveragedData(int numberOfDrugDataPerChannel, double \*\*dataPerChannel, double \*\*sortedDataPerChannel){

int iter,iter2;

double block,conc;

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

if(i==0){//first data of dataPerChannel

conc = dataPerChannel[i][0];//conc variable

block = dataPerChannel[i][1];//block variable

iter = 1;//counting number of same conc values

iter2 = 0;//row of sortedDataPerChannel

}

else{

if(dataPerChannel[i][0]==conc){

if(i<numberOfDrugDataPerChannel-1){

block = block + dataPerChannel[i][1];//adding block variables

iter = iter + 1;

}

else{//last data of dataPerChannel

block = block + dataPerChannel[i][1];//adding block variables

iter = iter + 1;

block = block/iter;

sortedDataPerChannel[iter2][0] = conc;

sortedDataPerChannel[iter2][1] = block;

//std::cout << i << " Sorted " << iter2 << " has value of conc = " << conc;

//std::cout << " and average block " << block << std::endl;

}

}

else{

block = block/iter;//average block

sortedDataPerChannel[iter2][0] = conc;

sortedDataPerChannel[iter2][1] = block;

//std::cout << i << " Sorted " << iter2 << " has value of conc = " << conc;

//std::cout << " and average block " << block << std::endl;

conc = dataPerChannel[i][0];//set new conc variable

block = dataPerChannel[i][1];//set new block variable

iter = 1;//restart counting same conc values

iter2 = iter2 + 1;//next row of sortedDataPerChannel

}

}

}

return;

}

The sorted data will be the stored in sortedDataPerChannel array.

## Code for linear and nonlinear LSM

As stated in Section **Error! Reference source not found.**, especially **Error! Reference source not found.** to **Error! Reference source not found.** that express the linear equation for linear LSM fitting, the implementation of LSM relies on ALGLIB C++where the main calculation is as follow:

* The minimum requirement to conduct the linear LSM fitting is declared first.

The algorithm requires that the experimental data have at least 4 different drug concentrations for each channel. It is implemented as:

if(numberOfDataLinFit<4){

std::cout << "Too few number of distinct conc values to fit Hill equation " << std::endl;

continue;

}

* If the minimum requirement is fulfilled, the next step is to calculate the actual initial guess of and . If the data is either dominated by very low or very high blocking, the and will be set as follow:

//criteria for linear fitting

if(flag1==numberOfDataLinFit){//all of the sortedDataPerChannell are very small blocking

//c0 = -log10(gsl\_vector\_max(dataConc));//log10(IC50)

c0 = -log10(\*std::max\_element(dataConc,dataConc+numberOfDataLinFit));//log10(IC50)

c1 = 1;//Hill coefficient

}

else if(flag2==numberOfDataLinFit){//all of the sortedDataPerChannell are very big blocking

c0 = -log10(\*std::min\_element(dataConc,dataConc+numberOfDataLinFit));//log10(IC50)

c1 = 1.0;//Hill coefficient

}

* If neither very low nor very high blocking dominate the data or ,in other words, we have a regular data, we can securely conduct linear LSM as follow:

//ignore data with dataFlag!=0

numberOfDataLinFit = numberOfDataLinFit - flag1 - flag2;

//std::cout << "This is the number of data to fit: " << numberOfDataLinFit << std::endl;

if(2 <= numberOfDataLinFit){

dataX = new double [numberOfDataLinFit];

dataY = new double [numberOfDataLinFit];

ignoreSomeData(numberOfDataLinFit,flag1,flag2,dataFlag,dataX,dataY,sortedDataPerChannel);

//initialize linear lsm environment

real\_1d\_array ylin;

real\_2d\_array fmatrix;

ylin.setlength(numberOfDataLinFit);

fmatrix.setlength(numberOfDataLinFit,2);

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

ylin[i] = dataY[i];

fmatrix[i][0] = 1;

fmatrix[i][1] = dataX[i];

}

int\_t infolin;

real\_1d\_array clin;

lsfitreport replin;

// Fitting without weights

lsfitlinear(ylin, fmatrix, infolin, clin, replin);

//printf("# best fit: Y = %g + %g X\n",clin[0],clin[1]);

//printf("# h = %g, log10IC50 = %g, IC50 = %g\n",clin[1],-clin[0]/clin[1],pow(10.0,-clin[0]/clin[1]));

c0 = clin[0];

c1 = clin[1];

delete [] dataX;

delete [] dataY;

Once the linear LSM finish, the initial and are further processed into nonlinear LSM fitting by following the procedure explained in Section **Error! Reference source not found.**. The implementation of nonlinear LSM fitting is done by using the following function:

void nonLinearFit(int numberOfDrugDataPerChannel, double c0, double c1, double \*log10IC50, double \*h, double \*\*dataPerChannel){

double conc;

real\_2d\_array x;

real\_1d\_array y;

x.setlength(numberOfDrugDataPerChannel,1);

y.setlength(numberOfDrugDataPerChannel);

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

x[i][0] = dataPerChannel[i][0];//drug concentration

y[i] = dataPerChannel[i][1];//blocking percentage

}

real\_1d\_array c;//parameters to fit

c.setlength(2);

//check the linear fitting results

//lower and upper bounds for h

if(c1 <= 0 || 10 <= c1){

//set new h parameter

c1 = 0.9;

}

//lower and upper bounds for log10IC50

if(-c0/c1 <= -10 || 10 <= -c0/c1){

//find average conc value

conc = 0.0;

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

conc = conc + dataPerChannel[i][0];//conc

}

conc = conc/numberOfDrugDataPerChannel;

//new log10IC50 = -c0/c1

c0 = -log10(conc)\*c1;

}

c[0] = -c0/c1;//log10IC50

c[1] = c1;//h

real\_1d\_array bndl;

bndl.setlength(2);

bndl[0] = -10.0;

bndl[1] = 0;

real\_1d\_array bndu;

bndu.setlength(2);

bndu[0] = 10.0;

bndu[1] = 10.0;

double epsx = 0.000001;

ae\_int\_t maxits = 0;

ae\_int\_t info;

lsfitstate state;

lsfitreport rep;

double diffstep = 0.0001;

lsfitcreatef(x, y, c, diffstep, state);

lsfitsetbc(state, bndl, bndu);

lsfitsetcond(state, epsx, maxits);

alglib::lsfitfit(state, function\_cx\_1\_func);

lsfitresults(state, info, c, rep);

//store results

\*log10IC50 = c[0];

\*h = c[1];

return;

}

This function will result the optimum log10IC50 and h.

## Code for Monte Carlo simulation

The implementation of Monte Carlo simulation is basically the same as the previous Appendix B except that it repeats the LSM fitting as many bootstrap samples as we need and it has different set of data on every iteration. There are two available schemes as in **Error! Reference source not found.**]. In the first scheme, the data is generated by using optimal or best-fit parameters from LSM fitting with Gaussian noise. The implementation of data generator for this scheme is as follow:

for(int j=0;j<numberOfDrugDataPerChannel;j++){

conc = dataPerChannel[j][0];//data of drug concentration

//block = dataPerChannel[j][1];//obtaining blocking from actual data

block = blocking(log10IC50\_Best,conc,h\_Best);//obtaining blocking from previous best-fit value

noise = gsl\_ran\_gaussian(r,SD);

dataPerChannelRand[j][0] = conc;//conc

if(block+noise<0){

dataPerChannelRand[j][1] = 0;//block

}

else{

dataPerChannelRand[j][1] = block+noise;//block

}

//printf("Conc: %f Blocking: %f Noise: %f \n",conc,block,noise);

}

The noise = gsl\_ran\_gaussian(r,SD) is given to the blocking data in every iteration. Each iteration will result different noise thus generates a different set of data. Meanwhile, the second scheme generates data from actual experimental data with Gaussian noise simply by changing the following expression:

block = blocking(log10IC50\_Best,conc,h\_Best);//obtaining blocking from previous best-fit value

to

block = dataPerChannel[j][1];//obtaining blocking from actual data

## Complete code for Hill fitting with LSM and Monte Carlo simulation

#include "stdafx.h"

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include "interpolation.h"

#include <iostream>

#include <iomanip>

#include <fstream>

#include <sstream>

#include <cstring>

#include <string>

#include <algorithm>

#include <math.h>

#include <gsl/gsl\_rng.h>

#include <gsl/gsl\_randist.h>

using namespace alglib;

//function for linear fitting

double Y(double block){

return log10(1.0/(1.0-block/100.0)-1.0);

}

double X(double conc){

return log10(conc);

}

double blocking(double log10IC50, double conc, double h){

return 100.0\*(1.0-1.0/(1.0+pow((conc/pow(10.0,log10IC50)),h)));

}

void function\_cx\_1\_func(const real\_1d\_array &c, const real\_1d\_array &x, double &func, void \*ptr);

void readData(std::string fileName, double \*\*data, std::string \*\*dataStr);

int readChannels(int numberOfData, std::string drugName, std::string\*\* dataStr);

int readDataPerChannel(int numberOfData, std::string drugName, std::string channelName, std::string\*\* dataStr);

void storeDataPerChannel(int numberOfData, std::string drugName, std::string channelName, std::string\*\* dataStr,double \*\*dataPerChannel, double \*\*data);

int readConcPerChannel(int numberOfDrugDataPerChannel, double \*\*dataPerChannel);

void storeChannelList(int numberOfData, std::string drugName, std::string\*\* dataStr, std::string\* channelList);

void findAveragedData(int numberOfDrugDataPerChannel, double \*\*dataPerChannel, double \*\*sortedDataPerChannel);

void ignoreSomeData(int numberOfDataLinFit, int flag1, int flag2, int\* dataFlag, double\* dataX, double\* dataY, double\*\* sortedDataPerChannel);

void nonLinearFit(int numberOfDrugDataPerChannel, double c0, double c1, double \*log10IC50, double \*h, double \*\*dataPerChannel);

double stdev(int numberOfDrugDataPerChannel, double log10IC50, double h, double \*\* dataPerChannel){

double result = 0.0, block, conc;

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

conc = dataPerChannel[i][0];

block = dataPerChannel[i][1];

result = result + pow(blocking(log10IC50,conc,h)-block,2.0);

}

result = sqrt(result/(numberOfDrugDataPerChannel-2));

return result;

}

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

{

//

// In this example we demonstrate Hill fitting by

// f(c,x)=100\*(1-1/(1+pow(x/pow(10,c0),c1)))

// subject to bound constraints

// -10.0 <= c0 <= 10.0

// 0.0 <= c1 <= 10.0

// using function value only.

// c0 = log10IC50 and c1 = h.

//

// Gradient is estimated using combination of numerical differences

// and secant updates. diffstep variable stores differentiation step

// (we have to tell algorithm what step to use).

//

//declaration for linear fitting

int numberOfData,numberOfChannels,numberOfDrugDataPerChannel;

int \*dataFlag,numberOfDataLinFit;

double conc,block,noise,eps=0.01;

double c0,c1,cov00,cov01,cov11,chisq;

double \*weight,SD,log10IC50,h;

double \*\*dataRaw,\*\*dataPerChannel,\*\*dataPerChannelRand,\*\*sortedDataPerChannel,\*\*finalSamples;

std::string \*\*dataStr,\*channelList,tempString,line,fileName,oFileName,bFileName,sFileName,headerSFile;

std::string drugName,channelName,drug,channel,units,pacing;

//initialization

//==============

if (argc != 2){

fprintf (stderr,"usage: ./lsm.out drugName\n");

exit (-1);

}

drugName = argv[1];

bFileName = "data/bootstrap\_"+drugName+".csv";

sFileName = "data/IC50\_samples\_"+drugName+".csv";

//open output file to store results

std::ofstream bFile(bFileName);

std::ofstream sFile(sFileName);

bFile << "channel,log10IC50,h" << std::endl;

//storing data from drug\_block.csv

//================================

fileName = "data/drug\_block\_validation.csv";

std::ifstream inputFile(fileName);

if(!inputFile.is\_open()){

std::cout << "Input drug\_block.csv file failed to open" << std::endl;

std::cout << "File name:" << fileName << std::endl;

std::cout << "==============================" << std::endl;

}

//skip file header

getline(inputFile,line);

//calculate number of drug data

numberOfData = 0;

while(getline(inputFile,line)){

numberOfData = numberOfData + 1;

}

inputFile.clear();

inputFile.seekg(0, inputFile.beg);

//preparing data array for conc and block

dataRaw = new double \* [numberOfData];

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

dataRaw[i] = new double [2];

}

//preparing an array for drug,channel,units and pacing

dataStr = new std::string\* [numberOfData];

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

dataStr[i] = new std::string [4];

}

//store inputFile into data and dataStr matrix

readData(fileName,dataRaw,dataStr);

//collecting name of channels

numberOfChannels = readChannels(numberOfData,drugName,dataStr);

//preparing an array for channel list

channelList = new std::string [numberOfChannels];

//storing channel names into channelList

storeChannelList(numberOfData,drugName,dataStr,channelList);

//preparing to store the finalSamples for sFile (drug IC50 bootstrap samples)

finalSamples = new double \*[2000];

for(int i=0;i<2000;i++){

finalSamples[i] = new double [2\*numberOfChannels];

}

//Start calculation value for all channels

//========================================

for(int channelNum=0;channelNum<numberOfChannels;channelNum++){

//collecting drug data per channel

channelName = channelList[channelNum];

headerSFile = headerSFile+channelName+"\_IC50,"+channelName+"\_h,";//Header for IC50 2,000 samples file

numberOfDrugDataPerChannel = readDataPerChannel(numberOfData,drugName,channelName,dataStr);

//storing dataPerChannel

dataPerChannel = new double \* [numberOfDrugDataPerChannel];

dataPerChannelRand = new double \* [numberOfDrugDataPerChannel];//For storing dataPerChannel with random noise

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

dataPerChannel[i] = new double [2];

dataPerChannelRand[i] = new double [2];

}

storeDataPerChannel(numberOfData,drugName,channelName,dataStr,dataPerChannel,dataRaw);

//find number of different conc values

numberOfDataLinFit = readConcPerChannel(numberOfDrugDataPerChannel,dataPerChannel);

std::cout << "Drug " << drugName << " with";

std::cout << " channel "<< channelName << " has ";

std::cout << numberOfDrugDataPerChannel << " number of data per channel";

std::cout << " and " << numberOfDataLinFit << " number of different conc values" << std::endl;

sortedDataPerChannel = new double \* [numberOfDataLinFit];

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

sortedDataPerChannel[i] = new double [2];

}

//finding average for sortedDataPerChannel

findAveragedData(numberOfDrugDataPerChannel,dataPerChannel,sortedDataPerChannel);

//linear fitting procedure

//========================

int flag1,flag2;

double \*dataX,\*dataY,\*dataConc;

//store initial dataFlag

dataFlag = new int [numberOfDataLinFit];

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

dataFlag[i] = 0;

}

//check the total number of sortedDataPerChannel

if(numberOfDataLinFit<4){

std::cout << "Too few number of distinct conc values to fit Hill equation " << std::endl;

continue;

}

else{

//counting flag1 for too many very small blocking

std::cout << "Count many very small blocking " << std::endl;

flag1 = 0;

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

if(sortedDataPerChannel[i][1]<eps){

flag1 = flag1 + 1;

dataFlag[i] = 1;

}

}

std::cout << "Number of too small blocking: " << flag2 << std::endl;

//counting flag2 for too many very big blocking

std::cout << "Count many very big blocking " << std::endl;

flag2 = 0;

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

if(sortedDataPerChannel[i][1]>100.0-eps){

flag2 = flag2 + 1;

dataFlag[i] = 2;

}

}

std::cout << "Number of too big blocking: " << flag2 << std::endl;

//copying conc data from sortedDataPerChannel

std::cout << "check" << std::endl;

dataConc = new double [numberOfDataLinFit];

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

dataConc[i] = sortedDataPerChannel[i][1];

}

//criteria for linear fitting

if(flag1==numberOfDataLinFit){//all of the sortedDataPerChannell are very small blocking

c0 = -log10(\*std::max\_element(dataConc,dataConc+numberOfDataLinFit));//log10(IC50)

c1 = 1;//Hill coefficient

std::cout << "check2" << std::endl;

}

else if(flag2==numberOfDataLinFit){//all of the sortedDataPerChannell are very big blocking

c0 = -log10(\*std::min\_element(dataConc,dataConc+numberOfDataLinFit));//log10(IC50)

c1 = 1.0;//Hill coefficient

std::cout << "check3" << std::endl;

}

else{

//ignore data with dataFlag!=0

numberOfDataLinFit = numberOfDataLinFit - flag1 - flag2;

std::cout << "This is the number of data to fit: " << numberOfDataLinFit << std::endl;

if(2 <= numberOfDataLinFit){

dataX = new double [numberOfDataLinFit];

dataY = new double [numberOfDataLinFit];

ignoreSomeData(numberOfDataLinFit,flag1,flag2,dataFlag,dataX,dataY,sortedDataPerChannel);

//initialize linear lsm environment

real\_1d\_array ylin;

real\_2d\_array fmatrix;

ylin.setlength(numberOfDataLinFit);

fmatrix.setlength(numberOfDataLinFit,2);

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

ylin[i] = dataY[i];

fmatrix[i][0] = 1;

fmatrix[i][1] = dataX[i];

}

ae\_int\_t infolin;

real\_1d\_array clin;

lsfitreport replin;

// Fitting without weights

lsfitlinear(ylin, fmatrix, infolin, clin, replin);

printf("# best fit: Y = %g + %g X\n",clin[0],clin[1]);

printf("# h = %g, log10IC50 = %g, IC50 = %g\n",clin[1],-clin[0]/clin[1],pow(10.0,-clin[0]/clin[1]));

c0 = clin[0];

c1 = clin[1];

delete [] dataX;

delete [] dataY;

}

else{

std::cout << "check4" << std::endl;

c1 = 0.9;//Hill coeff

//log10IC50 = -log10(conc)\*h

c0 = -log10(0.5\*(\*std::max\_element(dataConc,dataConc+numberOfDataLinFit)-\*std::min\_element(dataConc,dataConc+numberOfDataLinFit)))\*c1;

}

}

delete [] dataConc;

}//else

//Nonlinear lsm procedure

//=======================

double log10IC50\_Best, h\_Best,SD;

nonLinearFit(numberOfDrugDataPerChannel,c0,c1,&log10IC50\_Best,&h\_Best,dataPerChannel);

SD = stdev(numberOfDrugDataPerChannel,log10IC50\_Best,h\_Best,dataPerChannel);

printf("Best fit parameters from LSM:\n");

printf("log10IC50 = %f, h = %f \n",log10IC50\_Best,h\_Best);

printf("SD of residuals = %f\n",SD);

printf("=============================\n");

//Start Monte Carlo simulation to generate bootstrap

printf("Starting Monte Carlo simulation ...\n");

double \*MClog10IC50Samples,\*MChSamples;

MClog10IC50Samples = new double [2000];

MChSamples = new double [2000];

for(int MC=0;MC<2000;MC++){

//Modifying the block percentage on dataPerChannelRand with Gaussian noise

const gsl\_rng\_type \* M;

gsl\_rng \* r;

gsl\_rng\_env\_setup();

M = gsl\_rng\_default;

r = gsl\_rng\_alloc(M);

gsl\_rng\_set(r,clock());

for(int j=0;j<numberOfDrugDataPerChannel;j++){

conc = dataPerChannel[j][0];//data of drug concentration

//block = dataPerChannel[j][1];//obtaining blocking from actual data

block = blocking(log10IC50\_Best,conc,h\_Best);//obtaining blocking from previous best-fit value

noise = gsl\_ran\_gaussian(r,SD);

dataPerChannelRand[j][0] = conc;//conc

if(block+noise<0){

dataPerChannelRand[j][1] = 0;//block

}

else{

dataPerChannelRand[j][1] = block+noise;//block

}

//printf("Conc: %f Blocking: %f Noise: %f \n",conc,block,noise);

}

//find number of different conc values

numberOfDataLinFit = readConcPerChannel(numberOfDrugDataPerChannel,dataPerChannelRand);

//finding average for sortedDataPerChannel

findAveragedData(numberOfDrugDataPerChannel,dataPerChannelRand,sortedDataPerChannel);

//linear fitting procedure Monte Carlo

//====================================

int flag1,flag2;

double \*dataX,\*dataY,\*dataConc;

//store initial dataFlag

dataFlag = new int [numberOfDataLinFit];

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

dataFlag[i] = 0;

}

//check the total number of sortedDataPerChannel

if(numberOfDataLinFit<4){

std::cout << "Too few number of distinct conc values to fit Hill equation " << std::endl;

continue;

}

else{

//counting flag1 for too many very small blocking

flag1 = 0;

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

if(sortedDataPerChannel[i][1]<eps){

flag1 = flag1 + 1;

dataFlag[i] = 1;

}

}

//counting flag2 for too many very big blocking

flag2 = 0;

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

if(sortedDataPerChannel[i][1]>100.0-eps){

flag2 = flag2 + 1;

dataFlag[i] = 2;

}

}

//copying conc data from sortedDataPerChannel

dataConc = new double [numberOfDataLinFit];

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

dataConc[i] = sortedDataPerChannel[i][1];

}

//criteria for linear fitting

if(flag1==numberOfDataLinFit){//all of the sortedDataPerChannell are very small blocking

c0 = -log10(\*std::max\_element(dataConc,dataConc+numberOfDataLinFit));//log10(IC50)

c1 = 1;//Hill coefficient

}

else if(flag2==numberOfDataLinFit){//all of the sortedDataPerChannell are very big blocking

c0 = -log10(\*std::min\_element(dataConc,dataConc+numberOfDataLinFit));//log10(IC50)

c1 = 1.0;//Hill coefficient

}

else{

//ignore data with dataFlag!=0

numberOfDataLinFit = numberOfDataLinFit - flag1 - flag2;

if(2 <= numberOfDataLinFit){

dataX = new double [numberOfDataLinFit];

dataY = new double [numberOfDataLinFit];

ignoreSomeData(numberOfDataLinFit,flag1,flag2,dataFlag,dataX,dataY,sortedDataPerChannel);

//initialize linear lsm environment

real\_1d\_array ylin;

real\_2d\_array fmatrix;

ylin.setlength(numberOfDataLinFit);

fmatrix.setlength(numberOfDataLinFit,2);

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

ylin[i] = dataY[i];

fmatrix[i][0] = 1;

fmatrix[i][1] = dataX[i];

}

ae\_int\_t infolin;

real\_1d\_array clin;

lsfitreport replin;

// Fitting without weights

lsfitlinear(ylin, fmatrix, infolin, clin, replin);

c0 = clin[0];

c1 = clin[1];

delete [] dataX;

delete [] dataY;

}

else{

c1 = 0.9;//Hill coeff

//log10IC50 = -log10(conc)\*h

c0 = -log10(0.5\*(\*std::max\_element(dataConc,dataConc+numberOfDataLinFit)-\*std::min\_element(dataConc,dataConc+numberOfDataLinFit)))\*c1;

}

}

delete [] dataConc;

}//else

//nonLinear fitting procedure Monte Carlo

//=======================================

double log10IC50,h;

nonLinearFit(numberOfDrugDataPerChannel,c0,c1,&log10IC50,&h,dataPerChannelRand);

bFile << channelName << "," << log10IC50 << "," << h << std::endl;

finalSamples[MC][channelNum\*2] = pow(10,log10IC50);

finalSamples[MC][channelNum\*2+1] = h;

MClog10IC50Samples[MC] = log10IC50;

MChSamples[MC] = h;

gsl\_rng\_free(r);

}

//Sort the MC samples to obtain uncertainties (95% confidence intervals)

std::sort(MClog10IC50Samples,MClog10IC50Samples+2000);

std::cout << "Upper bound of 95% CI interval for log10IC50: " << MClog10IC50Samples[1899] << std::endl;

std::cout << "Lower bound of 95% CI interval for log10IC50: " << MClog10IC50Samples[99] << std::endl;

std::sort(MChSamples,MChSamples+2000);

std::cout << "Upper bound of 95% CI interval for h: " << MChSamples[1899] << std::endl;

std::cout << "Lower bound of 95% CI interval for h: " << MChSamples[99] << std::endl;

//Finish Monte Carlo simulation to generate bootstrap

printf("Monte Carlo simulation is finished...\n");

printf("=====================================\n");

delete [] MClog10IC50Samples;

delete [] MChSamples;

delete [] dataPerChannelRand;

delete [] dataPerChannel;

delete [] sortedDataPerChannel;

}//channel

//print sFile

sFile << headerSFile << std::endl;

for(int i=0;i<2000;i++){

for(int j=0;j<numberOfChannels;j++){

sFile << finalSamples[i][j\*2] << "," << finalSamples[i][j\*2+1] << ",";

}

sFile << std::endl;

}

delete [] finalSamples;

delete [] dataRaw;

delete [] dataStr;

delete [] channelList;

sFile.close();

bFile.close();

inputFile.close();

//outputFile.close();

return 0;

}

void function\_cx\_1\_func(const real\_1d\_array &c, const real\_1d\_array &x, double &func, void \*ptr)

{

// this callback calculates f(c,x)=100\*(1-1/(1+pow(x/pow(10,c0),c1)))

// where x is a drug concentration on X-axis and c is adjustable parameter

// c0 = log10IC50, c1 = h

func = 100\*(1-1/(1+pow(x[0]/pow(10,c[0]),c[1])));

}

void readData(std::string fileName, double \*\*data, std::string \*\*dataStr){

int iter;

double block,conc;

std::string line,tempString,drug,channel,units,pacing;

std::ifstream inputFile(fileName);

//skip file header

getline(inputFile,line);

iter = 0;

while(getline(inputFile,line)){

std::stringstream ss(line);

getline(ss,tempString,',');//drug

drug = tempString;

getline(ss,tempString,',');//conc

conc = atof(tempString.c\_str());

getline(ss,tempString,',');//units

units = tempString;

getline(ss,tempString,',');//channel

channel = tempString;

getline(ss,tempString,',');//block

block = atof(tempString.c\_str());

getline(ss,tempString,',');//pacing

pacing = tempString;

//store string and numerical data into arrays

data[iter][0] = conc;//store conc data

data[iter][1] = block;//store block data

dataStr[iter][0] = drug;//store dataStr

dataStr[iter][1] = units;//store dataStr

dataStr[iter][2] = channel;//store dataStr

dataStr[iter][3] = pacing;//store dataStr

iter = iter + 1;

}

inputFile.close();

return;

}

int readChannels(int numberOfData, std::string drugName, std::string\*\* dataStr){

int numberOfChannels;

std::string channelName;

numberOfChannels = 0;

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

if(dataStr[i][0]==drugName && numberOfChannels==0){

channelName = dataStr[i][2];//first channel

numberOfChannels = numberOfChannels + 1;

}

else if (dataStr[i][0]==drugName && numberOfChannels>0){

if(dataStr[i][2]!=channelName){

channelName = dataStr[i][2];//next channel

numberOfChannels = numberOfChannels + 1;

}

}

}

return numberOfChannels;

}

int readDataPerChannel(int numberOfData, std::string drugName, std::string channelName, std::string\*\* dataStr){

int iter;

iter = 0;

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

if(dataStr[i][0]==drugName && dataStr[i][2]==channelName){

iter = iter + 1;

}

}

return iter;

}

void storeDataPerChannel(int numberOfData, std::string drugName, std::string channelName, std::string\*\* dataStr,double \*\*dataPerChannel, double \*\*data){

int iter;

iter = 0;

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

if(dataStr[i][0]==drugName && dataStr[i][2]==channelName){//change channelList index for other channel

dataPerChannel[iter][0] = data[i][0];//conc

dataPerChannel[iter][1] = data[i][1];//block

iter = iter + 1;

}

}

return;

}

int readConcPerChannel(int numberOfDrugDataPerChannel, double \*\*dataPerChannel){

int iter;

double conc;

iter = 0;

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

if(i==0){

conc = dataPerChannel[i][0];//conc variable

iter = iter + 1;

}

else{

if(dataPerChannel[i][0]!=conc){

conc = dataPerChannel[i][0];//conc variable

iter = iter + 1;

}

}

}

return iter;

}

void storeChannelList(int numberOfData, std::string drugName, std::string\*\* dataStr, std::string\* channelList){

int iter;

std::string channelName;

iter = 0;

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

if(dataStr[i][0]==drugName && iter==0){

channelName = dataStr[i][2];//first channel

channelList[iter] = channelName;

iter = iter + 1;

}

else if (dataStr[i][0]==drugName && iter>0){

if(dataStr[i][2]!=channelName){

channelName = dataStr[i][2];//next channel

channelList[iter] = channelName;

iter = iter + 1;

}

}

}

return;

}

void findAveragedData(int numberOfDrugDataPerChannel, double \*\*dataPerChannel, double \*\*sortedDataPerChannel){

int iter,iter2;

double block,conc;

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

if(i==0){//first data of dataPerChannel

conc = dataPerChannel[i][0];//conc variable

block = dataPerChannel[i][1];//block variable

iter = 1;//counting number of same conc values

iter2 = 0;//row of sortedDataPerChannel

}

else{

if(dataPerChannel[i][0]==conc){

if(i<numberOfDrugDataPerChannel-1){

block = block + dataPerChannel[i][1];//adding block variables

iter = iter + 1;

}

else{//last data of dataPerChannel

block = block + dataPerChannel[i][1];//adding block variables

iter = iter + 1;

block = block/iter;

sortedDataPerChannel[iter2][0] = conc;

sortedDataPerChannel[iter2][1] = block;

}

}

else{

block = block/iter;//average block

sortedDataPerChannel[iter2][0] = conc;

sortedDataPerChannel[iter2][1] = block;

conc = dataPerChannel[i][0];//set new conc varibale

block = dataPerChannel[i][1];//set new block variable

iter = 1;//restart counting same conc values

iter2 = iter2 + 1;//next row of sortedDataPerChannel

}

}

}

return;

}

void ignoreSomeData(int numberOfDataLinFit, int flag1, int flag2, int\* dataFlag, double\* dataX, double\* dataY, double\*\* sortedDataPerChannel){

int iter;

iter = 0;

for(int i=0; i<numberOfDataLinFit+flag1+flag2;i++){

if(dataFlag[i]==0){

dataX[iter] = X(sortedDataPerChannel[i][0]);//conc

dataY[iter] = Y(sortedDataPerChannel[i][1]);//block

iter = iter + 1;

}

}

return;

}

void nonLinearFit(int numberOfDrugDataPerChannel, double c0, double c1, double \*log10IC50, double \*h, double \*\*dataPerChannel){

double conc;

real\_2d\_array x;

real\_1d\_array y;

x.setlength(numberOfDrugDataPerChannel,1);

y.setlength(numberOfDrugDataPerChannel);

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

x[i][0] = dataPerChannel[i][0];//drug concentration

y[i] = dataPerChannel[i][1];//blocking percentage

}

real\_1d\_array c;//parameters to fit

c.setlength(2);

//check the linear fitting results

//lower and upper bounds for h

if(c1 <= 0 || 10 <= c1){

//set new h parameter

c1 = 0.9;

}

//lower and upper bounds for log10IC50

if(-c0/c1 <= -10 || 10 <= -c0/c1){

//find average conc value

conc = 0.0;

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

conc = conc + dataPerChannel[i][0];//conc

}

conc = conc/numberOfDrugDataPerChannel;

//new log10IC50 = -c0/c1

c0 = -log10(conc)\*c1;

}

c[0] = -c0/c1;//log10IC50

c[1] = c1;//h

real\_1d\_array bndl;

bndl.setlength(2);

bndl[0] = -10.0;

bndl[1] = 0;

real\_1d\_array bndu;

bndu.setlength(2);

bndu[0] = 10.0;

bndu[1] = 10.0;

double epsx = 0.000001;

ae\_int\_t maxits = 0;

ae\_int\_t info;

lsfitstate state;

lsfitreport rep;

double diffstep = 0.0001;

lsfitcreatef(x, y, c, diffstep, state);

lsfitsetbc(state, bndl, bndu);

lsfitsetcond(state, epsx, maxits);

alglib::lsfitfit(state, function\_cx\_1\_func);

lsfitresults(state, info, c, rep);

//store results

\*log10IC50 = c[0];

\*h = c[1];

return;

}

|  |
| --- |
| Uncertainty Quantification through the Monte Carlo method for Parameter Estimation of In silico Model used for Drug Toxicity Assessment |
|  |
| Thesis for Master of Engineering |
| December 2020 |
| Ali Ikhsanul Qauli |