TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VAFA ANDALIBI

MODELING REALISTIC NEURONAL ACTIVITY IN MEA PLATES

Master of Science thesis

Supervisor: Dr. Francois Christophe
Examiner:Prof. Yevgeny Kucheryavy
Examiner and topic approved by the council of the faculty of Natural Sciences on October 7th 2015

ABSTRACT

VAFA ANDALIBI: MODELING REALISTIC NEURONAL ACTIVITY IN MEA PLATES
Tampere University of technology
Master of Science Thesis, 54 pages, 24 Appendix pages
November 2015
Master's Degree Program in Biomedical Engineering
Major: Medical Instrumentation – Communication Systems and Networks
Examiner: Professor Yevgeny Kucheryavy
Dr. François Christophe

Keywords: Computational models, Electrophysiology, MEA plate, Spiking Neural Networks, Bio-integrated Systems, Connectivity Analysis, Functional connectivity, Cox method, Causality.

This thesis work is part of a project from Academy of Finland aiming at integrating biological components in sensor networks. The current integration goal considers neuronal cultures for achieving data processing. Due to the high capacity of neuronal cultures in parallel computation, the main assumptions of this project are that such integration will enable data processing that is not achievable with electrical components, and will reduce energy consumption. Within the scope of this project, the objective of this thesis is to develop realistic computational models of neuronal cultures plated on Multi-Electrode Arrays (MEAs). MEAs are integrated circuits used for stimulating cell cultures and recording their electrophysiological activity. Such models are used in the project for feasibility simulations and preliminary developments of bio-integrated systems (BIS). The contribution of this thesis is twofold: modeling plausible neural cultures on MEA, and analysis of the connectivity of neural networks. The first part contributes in gaining an in-depth understanding of the behavior of the neural network in MEA plate. A simulation framework is designed, implemented and used to simulate the neuronal activity in a MEA plate containing 1000 neurons. Using the implemented framework, it is now possible to simulate a MEA plate with many customizable parameters, e.g. MEA size, neuron size, type and morphology. The second part contributes with two implementations of a method for functional analysis of neural networks. Two GPU-accelerated algorithms of the Cox method were implemented with the CUDA platform. The Cox method is a proven robust method for the analysis of functional connectivity in networks. This method, formerly demanding a long time as well as consequent CPU power, can now run hundreds of times faster on CUDA-supported GPUs in personal computers.

PREFACE

This Master of Science thesis had been done in the Department of Pervasive Computing in Tampere University of Technology, Tampere/Finland between November 2014 and the September 2015.

The purpose of this master thesis is twofold: first, introduce and demonstrate a new framework for simulating the behavior of biological neural network in MEA. Second, improve the performance of a robust method for connectivity analysis of neural networks, i.e. cox method, using a parallelization technique, i.e. CUDA programming. This dissertation will certainly benefit a wide range of researchers in the fields of neural networks, neuroscience as well as neurobiology labs, both by simulating the MEA plate and making it possible to run the Cox method, formerly demanding a long time as well as consequent CPU power, on CUDA-supported GPUs in personal computers.

I am very thankful to my supervisor Dr. Francois Christophe for having an open ear for my questions. He gave me professional advices and let me work on my own at the same time by evaluating my progress in working objectively. I also would like to say thank you to head of department of Pervasive Computing, Prof. Dr. Tommi Mikkonen for letting me to be part of his research group.

This research is funded by the Academy of Finland under project number 278882 – Bio-integrated Software Development for Adaptive Sensor Networks.

Tampere, 28.09.2015

Vafa Andalibi

CONTENTS

1. INTRODUCTION 1
1.1 Motivation 1
1.2 Contribution 2
1.3 Structure 3
2. BACKGROUND STUDIES 5
2.1 Behavioral Models of biological neurons 5
2.1.1 Hodgkin-Huxley model 5
2.1.2 Leaky-Integrate-and-Fire (LIF) model 5
2.1.3 Izhikevich Model 6
2.2 Models of neuronal plasticity 7
2.2.1 Hebb’s Rule 7
2.2.2 Oja’s Rule 8
2.2.3 Generalized Hebbian Algorithm (GHA) 8
2.2.4 Spike Timing Dependent Plasticity (STDP) 9
2.3 Connectivity Analysis 10
3. IMPLEMENTATION AND OUTPUTS 16
3.1 MEA Plate 16
3.1.1 Neuron 18
3.1.2 Synaptic Plasticity 18
3.1.3 Cell Distribution 19
3.1.4 Cell Death 20
3.1.5 Connections and Neuromorphology 21
3.1.6 Mini-Compiler 25
3.1.7 Cell Stimulation 26
3.1.8 Linking the Components 27
3.2 Functional Connectivity Analysis 28
3.2.1 Component Analysis of Cox method 28
3.2.2 1st algorithm 31
3.2.3 2nd algorithm 31
3.2.4 Test and performance evaluation of algorithms 33
4. RESULTS AND EVALUATION 34
4.1 MEA simulator 34
4.2 Functional Connectivity Analysis 36
4.2.1 CPU vs GPU 36
4.2.2 GPU alg. 1 vs GPU alg. 2 39
5. DISCUSSION 41
5.1 MEA Simulator 41
5.2 Connectivity Analysis 43
6. FUTURE WORKS 45
6.1 MEA simulator 45

7. CONCLUSION 47

7.1 Simulation of neuronal activity in MEA plate 47

7.2 Connectivity analysis of neuronal network 48

Appendix A1. Source Code of the Cox Method 55

Appendix A2. Cuda Kernel for Alg.1 58

Appendix A3. Cuda Kernel for Alg. 2 59

Appendix A4. Hessian Kernel 60

Appendix B. Inversed two-level diamond square alg. 61

Appendix C. Cell death 65

Appendix D. Morphology 66

LIST OF FIGURES

Figure 1. *Original Hodgkin-Huxley electrical circuit representing the differential equations relating membrane ionic currents and its electrical potential, reproduced from [28]* *6*

Figure 2. *An example for comparison between signals generated from models and signals recorded from biological neuron. (a) Illustration of comparison between data from a model and from 'in-vitro' experiment. (b) Comparison of short-term synaptic plasticity (STP) signal (in red) extracted from the model proposed in [33] with a signal of STP from in-vitro experiment (black signal with noise).* *7*

Figure 3. *Schematic of STDP drawn after 60 spike pairing from [42], regression curves reproduced using equation (8)* *9*

Figure 4. *Example of two connectivities that pairwise methods cannot differentiate but can be recognized by both ML estimation methods and Granger causality analysis, redrawn from [52]* *11*

Figure 5. *Simple spike train of three spikes and corresponding ISIs $x1, x2$ and $x3$.* *13*

Figure 6. *Spike train of Figure 5 sorted in ascending order based on length of ISIs and addressed as new values $x1, x2$ and $x(3)$. The $tx(i)x(j)$ values resulted by allocating smaller ISIs inside the larger ones ($i \geq j$), coinciding left ends and considering right end of $x(j)$ as the t value* *13*

Figure 7. *Example of adjacency matrix previously proposed in [23] as the final output of the cox method indicating both the connectivity map and the strength of the connection* *14*

Figure 8. *Original connection scheme proposed in [23], on which Cox method was applied. The length of spike trains recorded from each neuron was equal to 20,000 milliseconds* *15*

Figure 9. *The components considered in the model of MEA plate in current study* *16*

Figure 10. *Example of the firing pattern of a Regular Spiking (RS) pyramidal neuron simulated with Izhikevich's simple model [32]. (a) Simulation of the membrane potential response to a 70pA continuous excitation during 1s. (b) Phase portrait representation of the relations between the membrane potential v and the recovery current u.* *17*

Figure 11. *Example of the synaptic change as a result of synchronous firing. a) spike timing in a network of 25 neurons. b) voltage in two firing neurons c) synaptic weight* *20*

Figure 12. *Output of two-level inversed diamond-square algorithm used for determining the "allowed" location of the neurons in the MEA plate. Red pixels represent the locations that neurons can be placed on. a) 3D visualization of first level of algorithm b)2D visualization of the final result* *21*

Figure 13. *3D visualization of the death rate. a) after 3 DIV, b) after 9 DIV, c) after 17 DIV. Red pixels represent the locations that neurons can be placed on. Black pixels show the neurons previously located on red pixels that are currently died.* *21*

Figure 14. *3D Morphology of a rat's neocortex pyramidal cell extracted from NeuroMorpho database. In this morphology, the white color corresponds to soma (the small white dot in the upper middle of the figure), the gray and green colors are dedicated to axons and dendrites respectively. The magenta color adjacent to the soma illustrates the apical dendrite* *22*

Figure 15. *Morphology of the rat's neocortex pyramidal cell extracted from NeuroMorpho database. a) 3D representation of the morphology imported from swc file and shown using brian2 library. b) The same morphology converted in 2D and shifted 500 µm right and up in 2D coordinates* *23*

Figure 16. *Illustration of the difference between considering neuron as a) only a soma and b) a complete morphology in a simple MEA containing 4 neurons.* *23*

Figure 17. *Adding random rotation to each neuron for a more realistic connection between neurons* *24*

Figure 18. *The result of the connection_finder algorithm applied on a small network of 10 neurons with 7 pyramidal cells and 3 basket cells. Beginning from upper left figure, each plot is focused on part of the previous plot as indicated with a cyan square. In all four figures, the black and red lines represent axons and dendrites and the connection points are shown with green triangles.* *25*

Figure 19. *Multivariate Gaussian distribution utilized for simulating the effect of a single electrode usable in both recording from neurons and stimulating them. The blue and red color represent maximum and minimum contact between the electrode and adjacent neuron, respectively.* *26*

Figure 20. *Formation of the electrodes in a standard MEA simulated using multivariate Gaussian distribution. The corner electrodes are omitted since they are not used for recording and stimulating.* *27*

Figure 21. *Flow chart of the main tasks of the Cox method presented sequentially. The notations dL1 and dL2 in this chart correspond to the first and second differential of the log likelihood L according to* *the coefficients of connectivity, β. Thus, dL1 corresponds to the gradient of the log likelihood and dL2 corresponds to the Jacobian of this gradient, also called the Hessian matrix of the log likelihood.*.......30

Figure 22. *Simple network topology considered as example for describing the two GPU accelerated implementations of the Cox method.*.......31

Figure 23. *Schematic of the grid formation for the first algorithm where $Zref\ n(tij)$ indicates the Z value of reference neuron n at time tij*.......32

Figure 24. *Schematic of the grid formation for the first algorithm where $Zref\ n(tij)$ indicates the Z value of reference neuron n at time tij*.......32

Figure 25. *Simulation of the neuronal behavior in MEA plate previously presented in [57]*.......34

Figure 26. *Output of the current simulator*....... **Error! Bookmark not defined.**

Figure 27. *Output of Cox method in this study. a) Adjacency matrix of Cox coefficients b) Adjacency matrix of confidence intervals for Cox coefficients in a. The Cox coefficients are considered as a real connection if their corresponding confidence intervals does not contain zero. In such a situation, the value of confidence interval is taken as the connection strength. c) Adjacency matrix of connectivity result of the network. The values strength of each connection is extracted from the cox coefficient adjacency matrix.*.......37

Figure 28. *Performance (runtime) comparison of the CPU implementation of the Cox method with two GPU algorithms with number of neurons in the network as the variable.*.......38

Figure 29. *Performance (runtime) comparison of the CPU implementation of the Cox method with two GPU algorithms with length of spike train in the network as the variable.*.......39

Figure 30. *Comparison of execution times of between the 2 GPU implementations of the Cox method, as a variable of the number of neurons in the studied network.*.......40

Figure 31. *Comparison of execution times of between the 2 GPU implementations of the Cox method, as a variable of the average number of recorded spikes in the studied network.*.......40

LIST OF TABLES

Table 1. *Classification of methods for analysis of connectivity of neural networks..........10*
Table 2. *Distribution and types of the total of 493 neurons in the simulated MEA..........34*

LIST OF PROGRAMS

Program 1. *The code implemented in brian for Izhikevich neuron.* *18*
Program 2. *Implementation of STDP in brian between inhibitory and excitatory neurons.* *19*
Program 3. *The source code for the entire method* *57*
Program 4. *Kernel part (CUDA – part handled by GPU) of the code for the first algorithm* *58*
Program 5. *Kernel part (CUDA – part handled by GPU) of the code for the second algorithm.* *59*
Program 6. *Kernel for accelerating the computation of the Hessian of ML* *60*
This appendix, presents the code for inversed two-level diamond square algorithm. *61*
Program 7. *The code for inversed two-level diamond square algorithm used for plating* *64*
This appendix, presents the code for implementing cell death. *65*
Program 8. *Implementation of cell death* *65*
This appendix provides the code for generating the distributions based on morphology. *66*
Program 9. *The implementation used for utilizing morphology for the purpose of finding connections* *74*

LIST OF SYMBOLS AND ABBREVIATIONS

ANN	Artificial Neural Network
BIS	Bio-Integrated Systems
DIV	Days In Vitro
GHA	Generalized Hebbian Algorithm
GLM	Generalized Linear Model
ISI	Inter-Spike Intervals
JPTH	Joint Peristimulus Time Histogram
LIF	Leaky-Integrate-and-Fire
ML	Maximum of Likelihood
MSC	Mean-Square-Contingency
MEA	Micro-Electrode Array
MI	Mutual Information
RS	Regular Spiking
STP	Short-Term Plasticity
STDP	Spike Timing Dependent Plasticity
C	Constant parameter related to the capacitance of the membrane
V, v	Membrane Potential
I	Membrane Current
g_{leak}	Constant parameter leakage conductance of the membrane
E_{leak}	Constant parameter related to the leakage equilibrium potential of the membrane
k, a, b, c, d	Constant parameters in Izhikevich model
V_{reset}	Reset value of membrane potential
V_{thresh}	Threshold value of membrane potential
u	Membrane recovery current
α, γ	Learning rate
w_i	Connection strength of the synapse j between pre-synaptic and post-synaptic neurons
x_i	Pre-synaptic neuron
w_{ij}	connection strength between the ith input and the jth output
t^n	Firing time of the postsynaptic neuron at the time of spiking n
t_j^n	Firing time of the presynaptic neuron j at the time of spiking f
W(x)	STDP function
$\varphi_A(t)$	Proportional Hazard Function of Spike train of neuron A
$U_A(t)$	Duration since the last spike of neuron A
β_i	Representation for other neurons of the network affecting neuron
$Z_{B_i}(t)$	Influence function of β_i to A
L()	Log likelihood function

1. INTRODUCTION

1.1 Motivation

Integrating biological components as part of computer is a present-day challenge as it can improve the devices in matter of energy consumption whilst keeping their computational performance high [1], [2]. Researchers from various disciplines, e.g. robotics [3], [4] and communication networks [5], [6] are recently paying more attention to systems integrating biological parts [7]–[9]. An interesting application of these systems, named Bio-Integrated Systems (BISs), is a system controller interfaced with biological neuronal network cultured on an electrode grid surface. An example of these grids is Micro-Electrode Array (MEA) plate [10] that is capable of both stimulating the neurons and recording their firing patterns [9]. Advantage of neural interfacing is possibility of studying and manipulation of these networks which leads to clarification of their fundamental mechanisms. Furthermore, enhancement of current engineering functionalities as well as new ones would be made possible by interfacing with cell cultures [11].

Examples of the most recent stage in interface development of living culture with non-biological systems would be the closed-loop stimulus-response system developed by Potter et al. [11], the robot with biological brain developed in [3], Lego Mindstorm robot created by Shahf et al. [12], robot arm controlled with a biological neuronal network [13] and using living neurons to control the flight of a simulated aircraft [14]. In the work done by Shahaf et al. [12] the data is produced from ultrasonic sensors, aka “the eye” of the robot, and is used for stimulation of large random networks of neurons. In all of these examples, the vital component interfacing the non-biological system to biological neuronal network is MEA plate.

The early studies on field of MEA were mainly focused on development of MEA hardware. The rapid growth of utilization of these instruments in electrophysiology community has resulted in new applications. For instance, researchers used MEA to advance the study of hypothalamus, that is a key component of the brain regulating important body functions [15]. In other application, the hippocampus, that is a vital component in formation of memories and is the main research focus for Alzheimer treatment, is studied using MEA plate [16], [17]. Regarding the spinal cord studies and treatments, several researches employed MEAs [18], [19]. Other areas utilizing MEA are heart research [20], Hippocampal oscillation studies [21] and studies regarding the he synchronized activity of the cardiac muscles and stem cells [22]. MEA enables new types of experiments that cannot be maintained with traditional instrumentation.

Observing the ensembles of biological neural networks is essential for developing novel topologies of Artificial Neural Networks (ANNs). With this regard, functional analysis of neuronal connections [23] and connection changes [24] plays crucial roles due to two reasons. Firstly, causal relations between input stimuli and the activation of paths in a neural network becomes possible using this analysis. Secondly, strong relation between structure and functionality of a network can be built based on the same analysis. There is a hypothesis proposing a strong correlation between network's function and its structure. Based on this hypothesis, the analysis of the temporal connectivity between neurons can be utilized to reproduce neuronal network which was previously build for special functionalities such as face recognition, natural language processing or machine learning [25]–[27].

1.2 Contribution

Modeling and simulation is used to represent the behavior of a system, by reproducing the real experiments. The advantages of simulation and modeling are avoiding the expenses for building a prototype, damage prevention, easy modification and re-testing, and error detection and correction. In a real experiment, error cause fault and damage which will demands a correction and finally leads to a new prototype. The contribution of this thesis can be divided into two main sections. First, a realistic model of MEA plate and second, connectivity analysis of the data recorded from simulated or real biological neuronal network.

First, having a model for MEA plate benefits us in many aspects: saving lab expenses, difficulties in preparing cell cultures, preventing spontaneous failures due to human errors, re-performing an experiment with exact same condition, etc. This work is dedicated for implementing a model of MEA plate featuring as many realistic components as possible. The implementation of MEA plate in this work was implemented in python using the *brain2* library for simulation of spiking neural networks. This library is easy to use and comprehensible for developers. Moreover, by employing other sophisticated python libraries alongside with brian2, better performance and representation for results were achieved.

Second, in order to study the procedure of information processing in neuron groups and for understanding the neuronal interactions, a functional connectivity analysis method is demanded. Cox method, as a statistical technique for connectivity analysis that has been used in many previous studies, is capable of producing robust results. However, due to its statistical characteristic, this method is very heavy in matter of computation. By utilizing the parallel nature of graphics processing, this study improves the runtime speed of Cox method enormously. The second part of this work was implemented using PyCUDA, a wrapper of the CUDA API which provides the access to NVIDIA's CUDA parallel computation API from python, enabling execution of parallel computations on GPU.

These two main parts can be combined by inputting the simulation outputs resulted from the first part to the second part, i.e. Cox method. This Combining has two major advantageous. First, dynamic behavior of the connectivity in a neural network can be found and be compared to the known behaviors of a biological neural network. This can either strengthen or weaken the robustness of the simulation results and can be used as a tool to evaluate different behavioral models. Second, observing and controlling the connection of neurons in a neural network is the crucial part in the topic of supervised learning. Combination of the two main tools of this study, can lead to a more realistic simulation of supervised learning using spiking neural networks.

1.3 Structure

In order to have a realistic model of MEA plate, it should be broken down into its constitutive elements. Then, each element should be considered as a sub-system, necessitating a separate modeling and the seamless interaction of these sub-models should be handled as realistic as possible, whilst preserving the realistic characteristic. Based on these criteria, seven vital components of MEA plate are considered in this study: 1) neuron model 2) Neuronal interaction model 3) Stimulation 4) Distribution of neurons over the grid 5) Neuromorphology 6) Cell death 6) mini-compiler for inputting determined connection map to brian, cell stimulation. Beside above-listed components, there are two terms that are crucial for MEA when it is purposed to be employed as an interface in bio-integrated systems: Structural Connectivity and Functional Connectivity. These terms are also implemented and explained in details.

This thesis consists of major parts. In the second chapter, i.e. background studies, some well-known models of important components of a MEA plate, namely neurons and plasticity, as well as different methods for connectivity analysis of spiking neural networks is presented. The reviewed neuron models are Hodgkin-Huxley, Leaky-Integrate-and-Fire (LIF) model and Izhikevich model and for Neuronal Plasticity four famous plasticity rules, namely Hebb’s rule, Oja’s Rule, Generalized Hebbian Algorithm (GHA) and Spike Timing Dependent Plasticity (STDP) are presented. In section 2.3, a review of different connectivity analysis methods are provided and the selection of the best method is justified as well.

In chapter 3, first the internal components required for implementing a realistic simulation of MEA plate, i.e. neuron, synaptic plasticity, cell distribution, cell death, connections and neuromorphology, mini-compiler as well as the procedure of linking these components, is provided. In the second section, Cox method is firstly reviewed in short. This is followed by presenting two different algorithms for accelerating its performance using a parallel computing. Finally, the testing protocol the methods for its performance evaluation is provided.

Fourth chapter of this study provides the results from both the simulation of MEA plate after linking its internal components as well as a detailed performance evaluation and performance comparison of the Cox method. In the chapter 5 and 6, the discussion and potential future works of this study is presented.

2. BACKGROUND STUDIES

In the following section, works related to each component are reviewed in brief. This covers the main neuron models, main models of neuroplasticity, distribution algorithms, cell death and growth. Moreover, methods related to functional connectivity analysis of neuronal networks are reviewed as well.

2.1 Behavioral Models of biological neurons

As the essence of MEA plate, having a realistic computational model for biological neuron is vital. This model must at least represent the electrochemical behavior of neural cells as well as their growth. The former is modeled based on the flow of the ions as well as electrical current from outside to inside and vice versa. Regarding the neuronal connectivity, i.e. plasticity, the methods reviewed in following sections are based on synchronized activity of neurons. Inter-neuronal synapses strengthened as neurons fire synchronously.

2.1.1 Hodgkin-Huxley model

The accurate model proposed by Hodgkin and Huxley in 1952 [28], provides an electrical circuit as a representation of differential equations describing the cell membrane ionic currents and its electrical potential, as illustrated in Figure 1. Since this model features four differential equations, it is computationally heavy and is not proper for large scale simulations. Hence the later models focused on the dynamics of neurons in order to propose a computationally simpler model.

2.1.2 Leaky-Integrate-and-Fire (LIF) model

This model[29] replaces the Hodgkin-Huxley's four differential equations with one differential equation described in equation (1):

$$C\dot{V} = I - g_{leak}(V - E_{leak}) \tag{1}$$

where the membrane potential V is defined as a function of time and sum of ion gates currents, I. C, g_{leak} and E_{leak} represents the capacitance, leakage conductance and leakage equilibrium potential of the membrane respectively. When membrane potential reaches a threshold, the neuron is considered as firing an action potential. The drawback of this model is manual drawing of spikes when the membrane potential reaches the threshold.

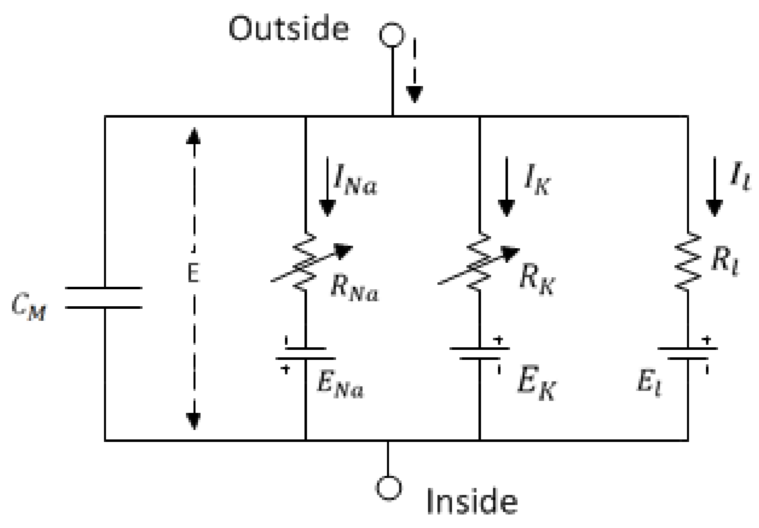


Figure 1. *Original Hodgkin-Huxley electrical circuit representing the differential equations relating membrane ionic currents and its electrical potential, reproduced from* [28]

2.1.3 Izhikevich Model

Later on, the issue in LIF model was addressed by Ermentrout [30], proposing a quadratic integrate-and-fire model. As opposed to LIF, this model was intrinsically capable of generating spikes and a dedicated value controls the peak value of the spike. In Hodgkin-Huxley model, there are many parameters that are dedicated to electrophysiological conductance. In practice, however, these parameters are difficult to measure. On the contrary, quadratic models possess fewer parameters which are easily adjustable in a way to match the real recordings [30].

Simple model of neuronal behavior proposed by Izhikevich [31], [32] can generate a very accurate dynamical behavior of neurons [24] whilst being computationally simpler than Hodgkin-Huxley model. This model is defined using following equations:

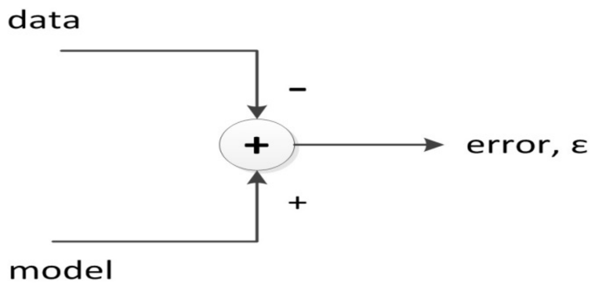
$$C\dot{v} = k(v - v_{rest})(v - v_{thresh}) - u + I, if\ v > v_{peak}, then\ v \leftarrow\ c, u \leftarrow u + d$$

$$\dot{u} = a[b(v - v_{rest}) - u] \qquad (2)$$

where $C, v, u, v_{rest}, v_{thresh}\ and\ v_{peak}$ represent membrane capacitance, membrane potential, recovery current, resting potential, threshold potential and peak potential respectively and k, a, b, c, d are constant parameters.

As a basic benchmark for neuron of Izhikevich, this model was compared with a signal recorded from a real experiment. There are multiple ways of such comparison, among which the simplest manner is to subtract one to the other and analyze the value of the resulting error. This practice, depicted in Figure 2 is usually used in closed-loop control.

a)



b)

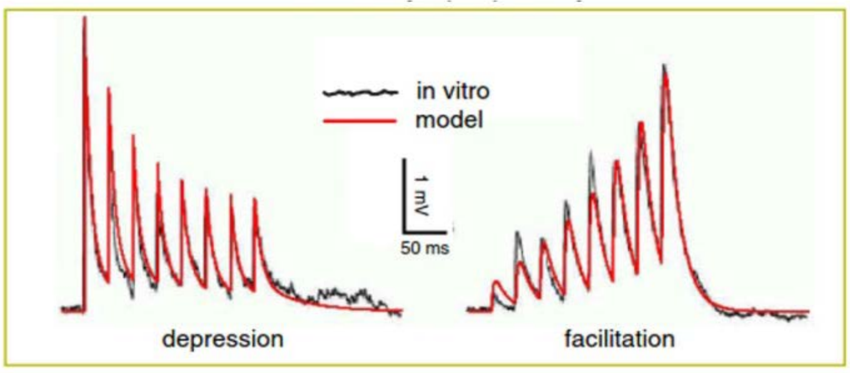


Figure 2. *An example for comparison between signals generated from models and signals recorded from biological neuron. (a) Illustration of comparison between data from a model and from 'in-vitro' experiment. (b) Comparison of short-term synaptic plasticity (STP) signal (in red) extracted from the model proposed in* [33] *with a signal of STP from in-vitro experiment (black signal with noise).*

As an example of this practice applied on Izhikevich model, Figure 2 illustrates the short-term plasticity (STP) in 'in-vitro' environment (black curves with noise). The red curve, which represents the generated STP model proposed in [33], is almost identical to the black curve. Nevertheless, determination of structure and function of a neuronal network necessitate an in-depth behavioral analysis of neurons during the network formation and solely modeling of firing patterns is not sufficient.

2.2 Models of neuronal plasticity

Neuronal plasticity is defined as the way with which neurons connect together and make their connection stronger.

2.2.1 Hebb's Rule

Regardless of its over-simplicity [34], the first model of plasticity, known as Hebb's rule [35], is the ground rule for modeling Spike-Timing Dependent Plasticity (STDP). In simple words, this models is expressed as "neurons that fire together, wire together" [36]. Hebb's rule is expressed as follows:

$$y = \sum_{i=1}^{n} w_i x_i \tag{3}$$

$$\Delta w_i = \alpha x_i y, i \in [\![1, n]\!] \tag{4}$$

Where w_i is the synaptic strength between pre-synaptic neuron x_i and post synaptic neuron y. However, based on to equation (4), the main issue in Hebb's rule is the fact that the connection can only grow in strength and it does not weaken in any circumstances.

2.2.2 Oja's Rule

In Oja's model of plasticity [37], the previous issue of Hebb's rule is solved. This was solved using a "forgetting" term $(-y^2 w_i)$. Oja's rule is expressed as following expression [38]:

$$\Delta w_i = \alpha(x_i y - y^2 w_i), i \in [\![1, n]\!] \tag{5}$$

where w_i is connection strength and Δw_i is the growth rate between pre-synaptic x_i and post synaptic y and α is learning rate. As an additional feature, this rule is capable of convergence testing by finding the covariance matrix of network connectivity with Oja's rule as principal component analyzer.

2.2.3 Generalized Hebbian Algorithm (GHA)

This model, proposed by Sanger [39], combines the Oja's rule and Gram-Schmidt process of orthogonalizing [40]. This model is presented in following equation:

$$\Delta w_{ij} = \gamma(t)(y_j x_i - y_j \sum_{k=1}^{j} w_{ik} y_k) \tag{6}$$

In equation (6) γ represents learning rate as a function of time, W_{ij} is connection strength between the ith input and the jth output. The conclusion of Sanger can be phrased as "*the weights converge to the eigenvectors of the input distribution*". As oppose Oja's rule which creates a neuronal network that converges to the first principal component, GHA is capable of finding eigenvector of a Principal Component Analysis [39].

2.2.4 Spike Timing Dependent Plasticity (STDP)

As the reference model of per- and post-synaptic connections. This model was proposed by Sjöström et al. [41] as follows:

$$\Delta w_j = \sum_{f=1}^{N} \sum_{n=1}^{N} W(t^n - t_j^f) \tag{7}$$

where Δw_j is strength changes in synapse j, calculated using sum of STDP function $W(t^n - t_j^f)$, in which W, t^n represents the *n*th spiking time of the postsynaptic neuron and *f*th spiking time of the presynaptic neuron j is represented as t_j^f. The function $W(x)$ is expressed as following form:

$$W(x) = \begin{cases} A_+ e^{-\frac{x}{\tau_+}}, & x > 0 \\ A_- e^{-\frac{x}{\tau_-}}, & x \leq 0 \end{cases} \tag{8}$$

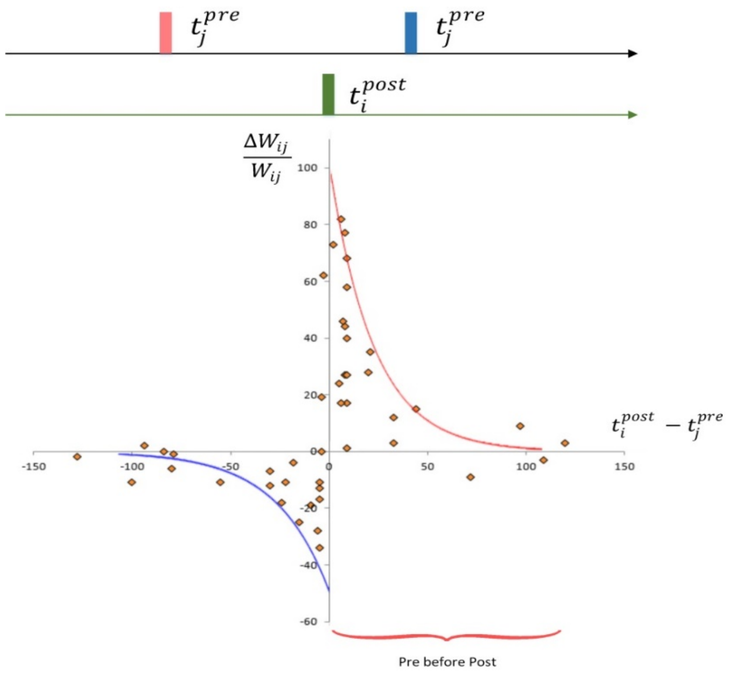


Figure 3. *Schematic of STDP drawn after 60 spike pairing from* [42], *regression curves reproduced using equation (8)*

This model is depicted in Figure 3, where positive growth of synaptic connection is happened due to pre-synaptic neuron activity. This effect can be presented as a negative strength when the pre-synaptic spikes arrives after post-synaptic spike.

2.3 Connectivity Analysis

Generally, connectivity analysis methods can be categorized based on four criteria: whether or not a method is statistical, pairwise, Binless or real-time. This classification is illustrated in Table 1. In [43], Functional connectivity methods are expressed in two main categories: phase synchronization and statistical measures. This division is integrated as the first criterion in Table 1. There are two steps in synchronization analysis: instantaneous phase estimation and quantification of phase locking. PHs methods, however, are not applicable for BISs as it is based on deterministic dynamical system principle.

Table 1. *Classification of methods for analysis of connectivity of neural networks*

		Criteria			
		Statistical	Non-pairwise	Binless	Real-time
Methods	Kalman Filter	X	X	X	X
	Cox	X	X	X	
	CuBIC	X	X	X	
	GLM	X	X		
	MI	X		X	
	MSC	X		X	
	Cross-correlation				
	Phase-Synchro.				
	Instant phase estimation				

Pair-wise comparison methods of connectivity analysis are being widely used, e.g. cross correlation method. Mutual Information (MI) [44] and Mean-Square-Contingency (MSC) [45] can be used to quantify statistical dependencies. These methods utilize the statistical information of the joint space between two random variables. The major problem with these category is that they only consider the influence from the pair of trains being studied. However, since a single spike is affected by many more factors than a spike train from one post-synaptic neuron [46], the methods that rely on the pair-wise analysis of spike trains are not sufficient to characterize the connectivity of neuronal network.

The essence of the pair-wise methods for analysis of dependencies between spike trains, e.g. CCF [47], cross-intensity function [48], method of moments [49], coherence calculation [50] and joint peristimulus time histogram (JPSTH) [51], is to compare the spike trains pair by pair. For this reason, these methods have a main weakness of not observing the influence of all neurons in the network and therefore cannot distinguish the direct and indirect connectivity between the nodes as depicted in Figure 4.

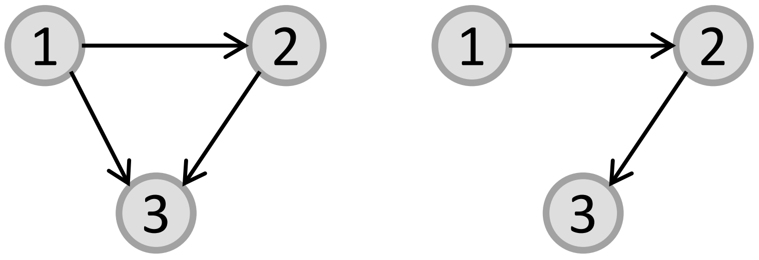


Figure 4. *Example of two connectivities that pairwise methods cannot differentiate but can be recognized by both ML estimation methods and Granger causality analysis, redrawn from* [52]

Regarding the next criterion, i.e. bin-dependency, this group of methods study the probability of appearance of a spike resulting from influence of all other spike trains and even its own previous activity. This probability is evaluated by calculating the Maximum of Likelihood function (ML). Generalized Linear Model (GLM) is among these group of methods taking into account all these influences and is applied to different cases of connectivity analysis. The best practice for functional connectivity is deduced to be using multiple time scales. In this case, the results would be highly dependent on the testing window (bin) [53].

The preferences for selecting a proper method is to select among binless ones as they are sensitive even with small amount of data. CuBIC method [54], is a successful attempt to suppress the effect of the bin in the computation of higher order correlations. The need for higher-order computation is estimated and unnecessary high-order computation is bypassed.

As for the final criterion, based on the time scale of 'in-vitro' experiment protocols and purpose of BIS development, the real-time connectivity analysis of a neural network with methods such as Kalman Filter [55] is not crucial.

Based on granger causality, for two signals X_1 and X_2 where the former "Granger-causes" the latter, a better prediction of X_2 can be achieved using the past values of X_1 compared to the information contained in past values of X_2 alone [52]. For instance, suppose we have three terms X_t, Y_t and W_t and the goal is to predict the value of X_{t+1}. If better prediction is achieved using the past terms of all three variables compared to using only X_t and Y_t, it can be said that past values of W_t contain helpful information in forecasting

X_{t+1} which cannot be found in Y_t and X_t itself. Therefore, W_t would "Granger cause" X_{t+1} in two conditions [52]:

- The occurrence of W_t is before X_{t+1}
- The information found in W_t is useful for predicting the value of X_{t+1} and cannot be found in Y_t

Based on the aforementioned characteristics, many of the common methods of connectivity analysis were classified and reviewed in [56]. The Cox method as a non-pairwise, statistical and binless method is known as a robust method for offline study of network connectivity. This method was initially proposed in signal processing area with the purpose of analyzing the multivariate point processes. In [23], Borisyuk et al. showed that such analysis is proper for analyzing the signal recorded from spiking neural network. Cox method is built on the assumption that a spike on a spike train is modulated by other trains produced by other neurons of the network, i.e. the modulated renewal process (MRP). The model of this MRP is presented as a hazard function expressing the probability of a spike rate at time t relatively to all inter-spike intervals (ISIs) of a spike train of length t or more [23], [55]. In this model, the proportional hazard function for a spike train recorded from neuron A, in a network with n+1 neurons is formulated as:

$$\varphi_A(t) = \varphi_A(U_A(t)).\, e^{\sum_{i=1}^{n} \beta_i Z_{B_i}(t)} \qquad (9)$$

where $\varphi_A(t)$ represents the proportional hazard function of spike train of neuron A, $U_A(t)$ is the duration since the last spike of neuron A, β_i is a representation for other neurons of the network affecting neuron A, and $Z_{B_i}(t)$ is the influence function of β_i to A. The computation of the log likelihood function is required for estimating the exponential part of the equation (9). This log likelihood function is expressed as following equation:

$$L(\vec{\beta}) = \sum_{i=1}^{n}\sum_{k=1}^{m} \beta_i . Z_{B_i}(t_{kk}) - \sum_{k=1}^{m} \log\left\{\sum_{l=k}^{m} \exp\left(\sum_{i=1}^{n} \beta_i . Z_{B_i}(t_{lk})\right)\right\} \qquad (10)$$

where *n* and *m* are the number of neurons that possibility can have effect on the target and number of recorded spikes respectively. With a spike train sorted based on the length of its ISIs and ∀k < l (l and k are indices of an ISI), t_{lk} is calculated as right end of the kth ISI whilst it is inserted inside the lth ISI in a way that their left ends coincide. This sorting process is presented with the example of a spike train of 3 spikes as in Figure 5, for which ISIs are sorted in Figure 6. Equation (10) is build based on the fact that the shortest ISIs between spikes has the highest influence on the target spike.

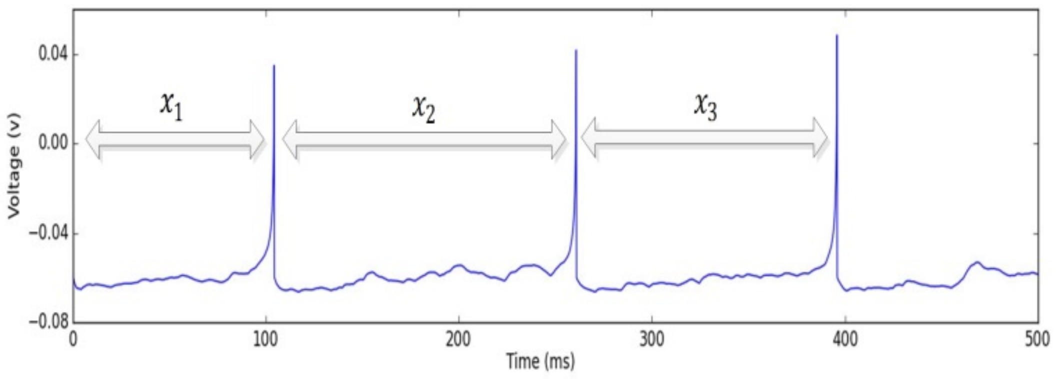


Figure 5. *Simple spike train of three spikes and corresponding ISIs x_1, x_2 and x_3.*

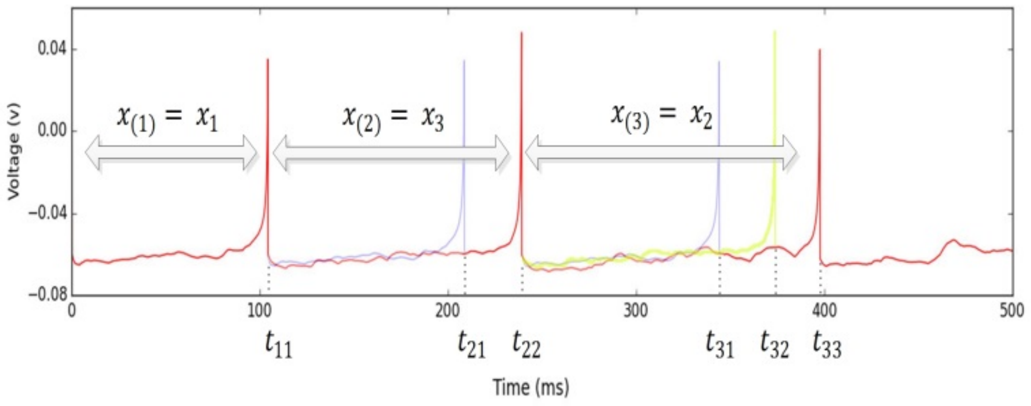


Figure 6. *Spike train of Figure 5 sorted in ascending order based on length of ISIs and addressed as new values $x_{(1)}, x_{(2)}$ and $x_{(3)}$. The $t_{x_{(i)}x_{(j)}}$ values resulted by allocating smaller ISIs inside the larger ones ($i \geq j$), coinciding left ends and considering right end of $x_{(j)}$ as the t value.*

Eventually, connectivity analysis will provide us with the results in form of adjacency matrix, as will be presented and described in section 4.2 in which cox method is applied on a simple network of 5 neurons. This adjacency matrix consists of three parts: 1) adjacency matrix of beta values, 2) adjacency matrix of confidence interval for beta values and 3) final connectivity results. In the first part, the value of Cox coefficients for each neuron is presented. Due to statistical nature of Cox method, these values are measured statistically and cannot be relied on without considering the Confidence Intervals which are provided in the second part. Based on the confidence intervals which are provided in this part, it is possible to find out which values in first table are reliable: if the confidence interval range contains zero, the cox-coefficient value is not a reliable once, hence should be removed from final results.

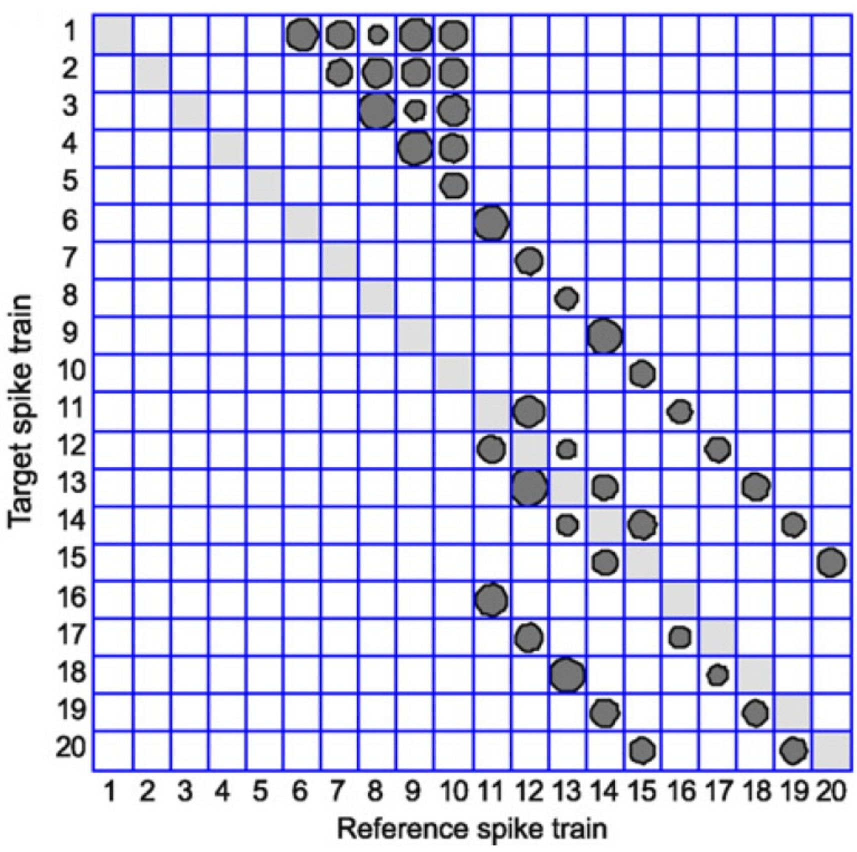


Figure 7. *Example of adjacency matrix previously proposed in [23] as the final output of the cox method indicating both the connectivity map and the strength of the connection.*

In other words, if the confidence interval values of a Cox coefficient value are both either positive or negative, the Cox coefficient value is acceptable, otherwise it should be omitted.The connectivity result can be represented in the form of a connection scheme as is presented in Figure 7. This figure, is built after applying cox method on the spike trains recorded from neurons with the original connectivity scheme provided in Figure 8. In in Figure 7, not only the connection map of the neurons are observable, the strength of each connection can be understood based on the area of its circle.

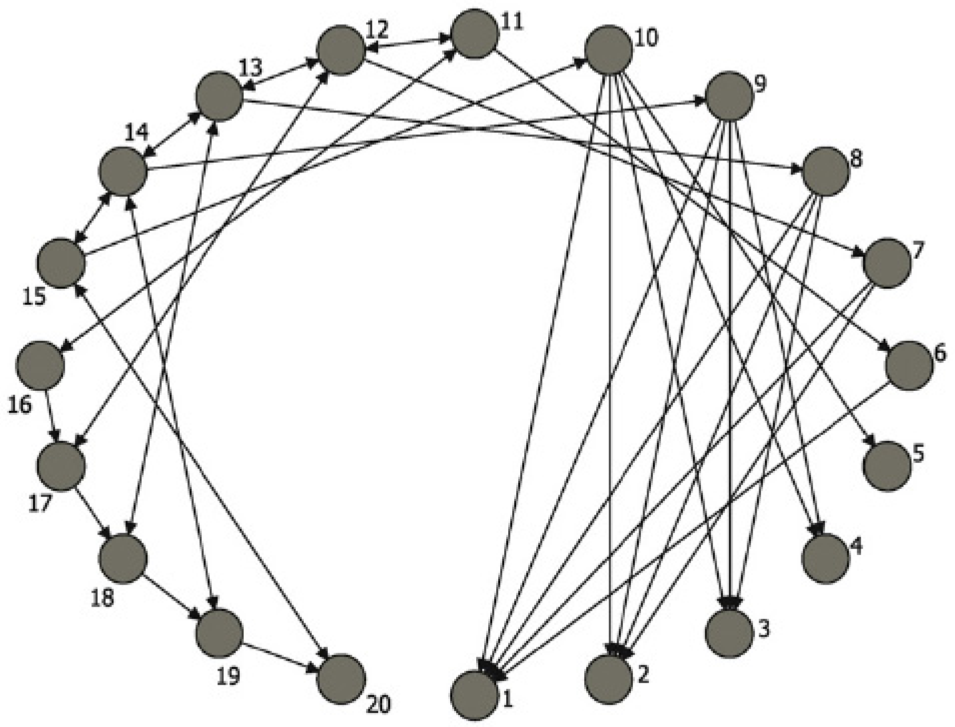


Figure 8. *Original connection scheme proposed in [23], on which Cox method was applied. The length of spike trains recorded from each neuron was equal to 20,000 milliseconds.*

3. IMPLEMENTATION AND OUTPUTS

The description of methodologies in this section consists of two main parts: in the first section, the components of the MEA plate are described and in the second section, the components for functional connectivity analysis are presented. Note that in each section, for better understanding of the implementation and methodology approaches, the partial output related to that sub-section is presented in this section and in the result section, the final output resulted from the whole system as a combination of all components will be produced.

3.1 MEA Plate

The model of MEA plate contains eight components, each of which simulates a vital part of the MEA plate: the morphology, distribution of the cells, cell death, stimulator, mini-compiler for *brian* and neuron and neuroplasticity models in *brian*.

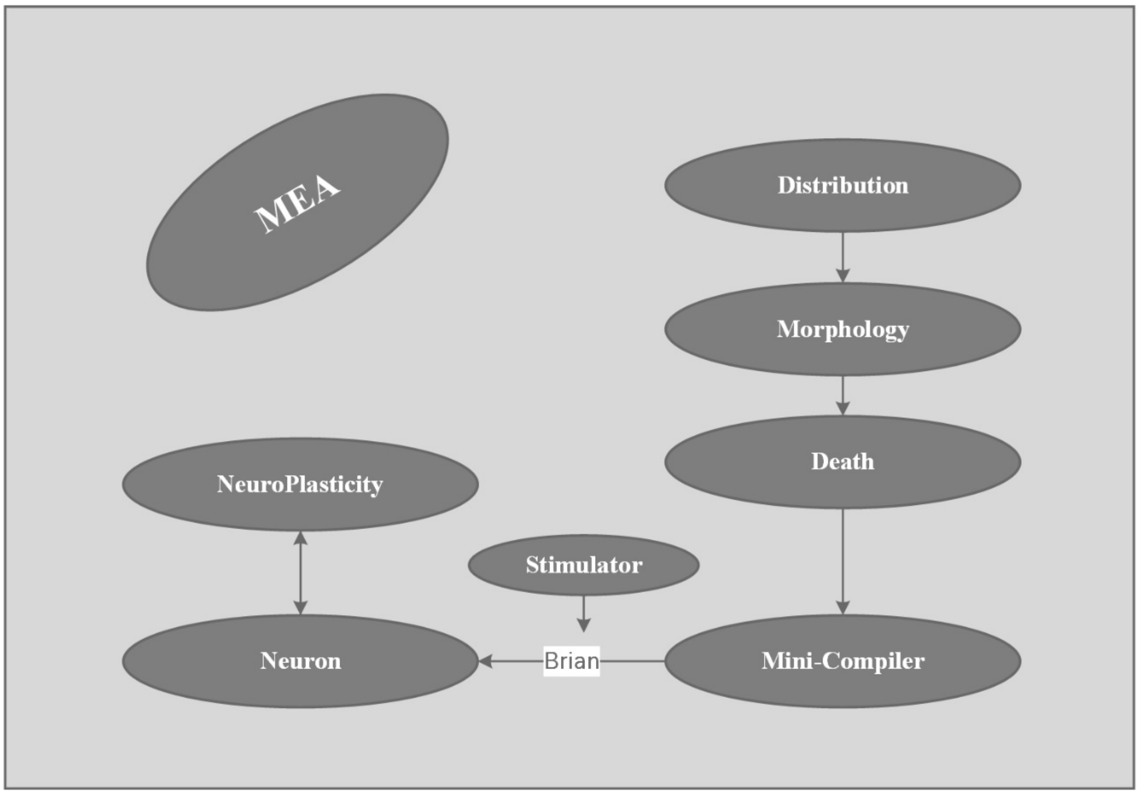


Figure 9. *The components considered in the model of MEA plate in current study.*

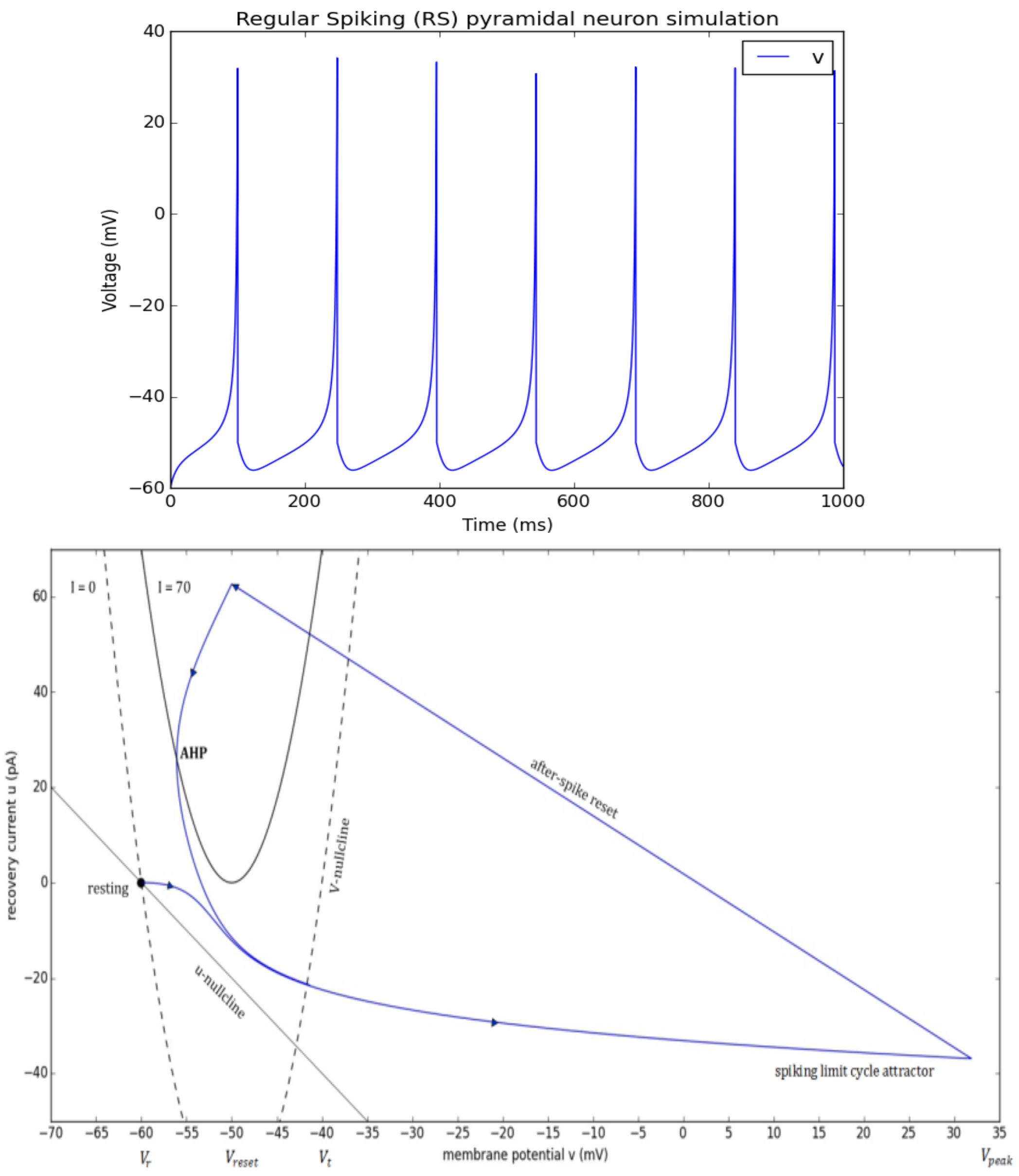


Figure 10. *Example of the firing pattern of a Regular Spiking (RS) pyramidal neuron simulated with Izhikevich's simple model* [32]. *(a) Simulation of the membrane potential response to a 70pA continuous excitation during 1s. (b) Phase portrait representation of the relations between the membrane potential v and the recovery current u.*

The approach taken here is bottom-up approach. As can be seen in figure (8) the idea of this approach begins with testing the model of single neuron. In the next step of this approach, the model is extend with neuronal plasticity, connectivity between the neurons and their growth. Finally, the model and its components are scaled up to full network of neurons with help of a cells distribution and evaluation of death rate during the process.

Regarding the provided components in figure (8), the intra-component relations are represented in the form of arrows. After determining the distribution of the neurons and using the source morphology file downloaded from neuromorpho library, the location of each point of cells are determined. In the next step, the death rate is applied on the neural population. The final results are compiled to be usable in *brian*. This compilation is maintained by generating syntaxes and running them inside *brian*, hence the link between mini-compiler and *brian*. In *brian*, the effect of stimulator is applied on both recording from neurons and stimulating them with current. Finally, the neuron model as well as plasticity, which is inextricably linked with the neuron model, are used inside *brian*.

3.1.1 Neuron

As described in section 2.1, there are many representations of neurons. Izhikevich model of spiking neuron was selected and implemented for this component. As oppose to Hodgkin-Huxley model which has a representation of neuronal components, this model reproduce the neuronal behavior mathematically. Whilst having a rational computational complexity, Izhikevich model produce precise results very close to real neurons. In *brian* the Izhikevich equation was reproduced using the following part of code:

```
eqs_neurons = '''
dv/dt=(k*(v-vr)*(v-vt)-u)/memc + (-ge*v - gi*(v-er)+ external)/memc :
volt
du/dt=a*(b*(v-vr)-u) : amp
dge/dt = -ge / taue : siemens
dgi/dt = -gi/taui : siemens
'''
reset = '''
v=c
u+=d
'''
```

Program 1. *The code implemented in brian for Izhikevich neuron.*

In Program 1, the first variable, i.e. *eqs_neurons*, represents the two main formula of Izhikevich and the other variable, *reset*, defines the reset event of that mode. The parameters *ge* and *gi* in the first variable are used for applying the excitatory and inhibitory synapses on the neuron. Figure 10 shows an example of dynamical behavior of the Izhikevich model. In this example, the firing pattern of a Regular Spiking (RS) pyramidal neuron simulated based on Izhikevich simple model [32] is illustrated. Moreover, the phase portrait representation of the membrane potential *v* and the recovery current *u* association is depicted.

3.1.2 Synaptic Plasticity

The implementations of this study are based on STDP model which was illustrated in Figure 3. The strength of the connection between neurons changes as a result of spiking in previous neuron. The formation of connections between neurons are considered as a separate component in this study, described in section 3.1.5. Figure 11, shows the changes of the synaptic weights between two groups of neurons.

The presynaptic and post synaptic events are the most important variables for implementing the STDP plasticity. These two variables are illustrated in Program 2.

```
eqs_stdp_inhib = '''w : 1
                dA_pre/dt = -A_pre / tau_stdp : 1 (event-driven)
                dA_post/dt = -A_post / tau_stdp : 1 (event-driven)'''
pre_in_ex = '''A_pre += 1.
               w = clip(w+(A_post-alpha)*eta, 0, gmax)
               gi += w*nS'''
post_in_ex = '''A_post += 1.
                w = clip(w+A_pre*eta, 0, gmax)
                '''
S10 = Synapses (Pi,Pe, model= eqs_stdp_inhib,
                   pre = pre_in_ex ,
                   post = post_in_ex)
```

Program 2. *Implementation of STDP in brian between inhibitory and excitatory neurons.*

3.1.3 Cell Distribution

In a real MEA plate, the way cells are distributed over the plate, has a strong effect on the formation of connection. In previous study maintained by Shultz [57], midpoint displacement fractal algorithm [58] was selected for cell distribution. In this study, however, a new algorithm is proposed to reproduce a more realistic distribution of the neurons in MEA plate. This algorithm is entitled *two-level inversed diamond-square algorithm*.

The core of this algorithm is the successive algorithm of midpoint displacement fractal algorithm proposed by Miller et al. [59], called *diamond-square algorithm*. However, the improved distribution still suffers from accumulation of the cells across the edges of the MEA plate. To address this issue, an *inversion* of the result produced by algorithm is considered, providing a sensible distribution. In other word, the random places which were generated with the algorithm were considered as the prohibited places and the original prohibited places were used as the result of algorithm. Though, after using the inversion, the neurons would still not be distributed evenly all over the MEA plate and will only concentrated in specific parts. The reason is the broad difference between the area of a single some, which is about $50\mu m^2$, and area of the whole MEA plate which is 7,480,000 μm^2.

To compensate this enormous difference, the plating is implemented in *two-levels*. First a space of 5000 μm^2 is preserved for each neuron. Then the diamond-square algorithm is applied for determining the location of the MEA that neurons with large area of 5000 μm^2 are allowed to be placed on. At this stage, however, the possible space for soma of each neurons is 100 times greater than its real size. In the next step, the exact location of each soma, inside its square of area of 5000 μm^2 is determined. This hybrid algorithm is called two-level inversed diamond-square algorithm from now on in this document. Apart from this algorithm, the resulted distribution is visualized in an interactive 3D image. An example of the visualized MEA plate with an area of 7,480,000 μm^2 containing at least 1000 cells each with the area of 5000 μm^2 is illustrated in Figure 12.

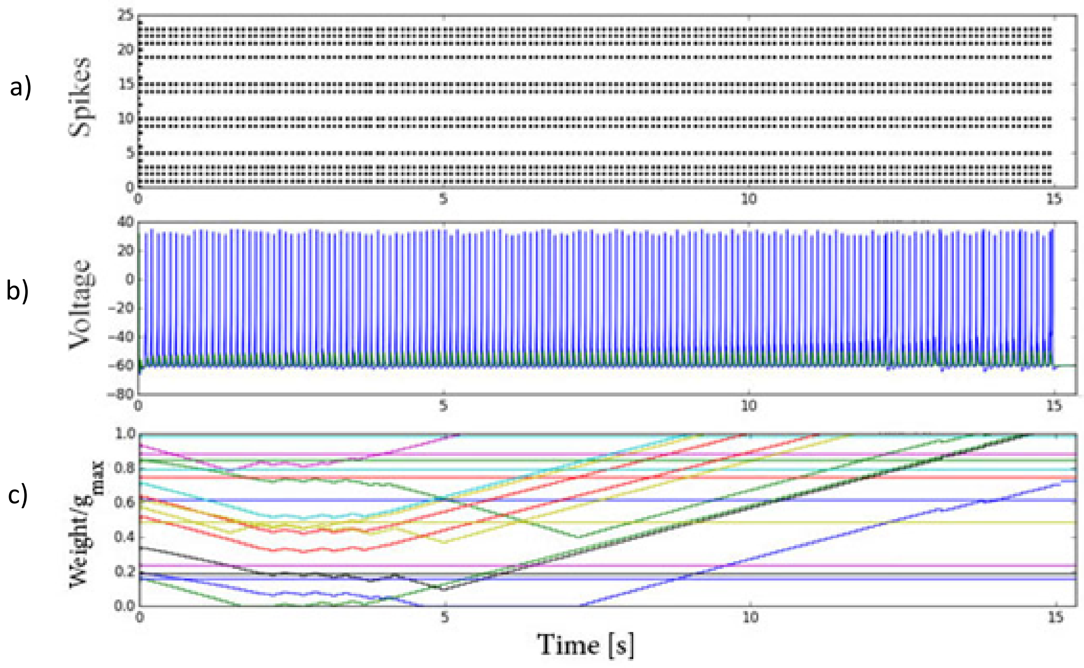


Figure 11. *Example of the synaptic change as a result of synchronous firing. a) spike timing in a network of 25 neurons. b) voltage in two firing neurons c) synaptic weight*

3.1.4 Cell Death

Before the final formation of the network and in the first 17 Days In Vitro (DIV), up to 60% of the cells die [60]. This death of the cultured neurons can have a marked effect on the other cells in the MEA plate. In the study presented by Shultz, this death was simulated by removing 45-60% of the cells before starting the main simulation. Using this method, however, the effect of the cells which are supposed to die in the 17 first days, are not taken into effect. A more precise method would be to wipe out the effect of the dead cells at the time of their death. Using this method, the possible effects of a neuron on the whole system before its death, will not be overlooked. The implementation of this part is maintained by 2D random sampling over the minutes of the days and number of neurons.

Using this method, the time of death is determined with a precision of minute. Cell death progression is also visualized in 3D view as in Figure 13. In this figure, the death rate of cells are depicted after 3, 9 and 17 days in vitro.

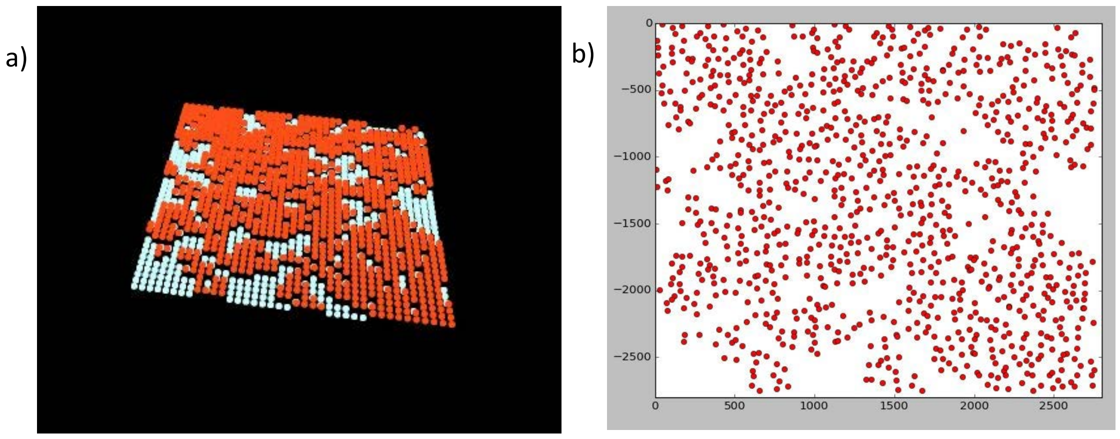


Figure 12. Output of two-level inversed diamond-square algorithm used for determining the "allowed" location of the neurons in the MEA plate. Red pixels represent the locations that neurons can be placed on. a) 3D visualization of first level of algorithm b)2D visualization of the final result

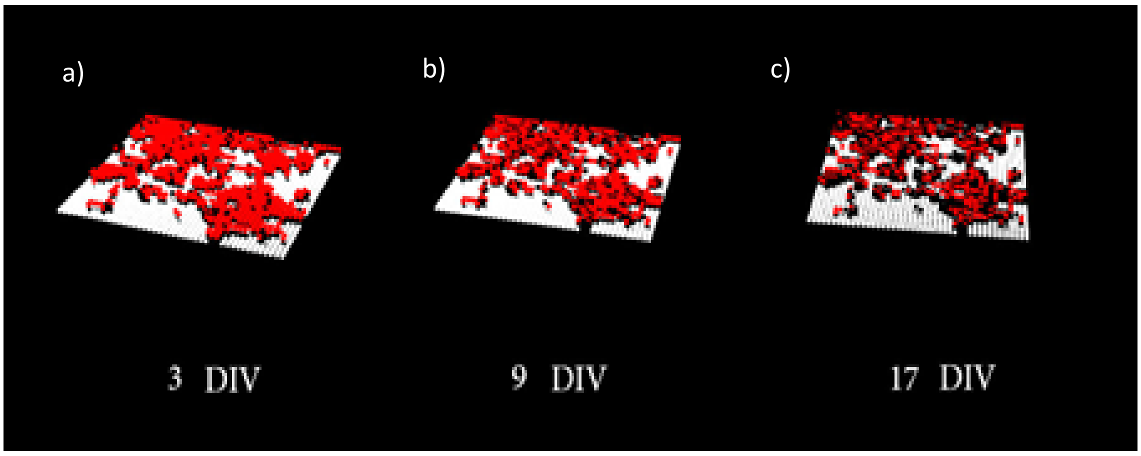


Figure 13. 3D visualization of the death rate. a) after 3 DIV, b) after 9 DIV, c) after 17 DIV. Red pixels represent the locations that neurons can be placed on. Black pixels show the neurons previously located on red pixels that are currently died.

3.1.5 Connections and Neuromorphology

Neuromorphology is defined as the study of the form and structure of the neurons. Have this concept not considered in any neuronal simulation, the result will suffer from many loopholes. This loopholes are mainly caused by the fact that neurons are considered solely as a square or circular soma. However, a realistic modeling of connection formation between neurons should be at the minimum based on the position of axons and dendrites. Figure 14, illustrates a morphology of a rat's neocortex pyramidal cell from the NeuroMorpho database located at www.neuromorpho.org. The morphologies are provided in *swc* format which is simple to understand and easy to modify.

As will be thoroughly discussed in section 5 major difference between this work and previous studies is taking the neuromorphology into account. It is of significant that the morphologies downloaded from aforementioned database are in 3D format. In MEA plate, however, the cells are connected to each other in a flat structure. In other word, cells will not grow connection vertically. For this reason, the 3D morphology is in the first step converted to a 2D one. Figure 15 shows the same morphology of Figure 14 imported in the simulator, converted to a 2D structure, shifted right and above, and showed using the brian2 library.

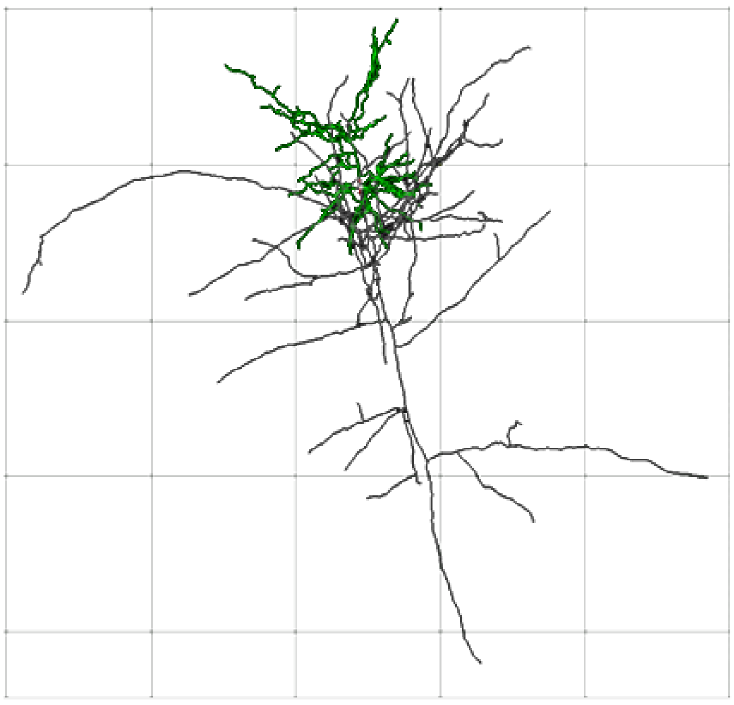


Figure 14. *3D Morphology of a rat's neocortex pyramidal cell extracted from NeuroMorpho database. In this morphology, the white color corresponds to soma (the small white dot in the upper middle of the figure), the gray and green colors are dedicated to axons and dendrites respectively. The magenta color adjacent to the soma illustrates the apical dendrite*

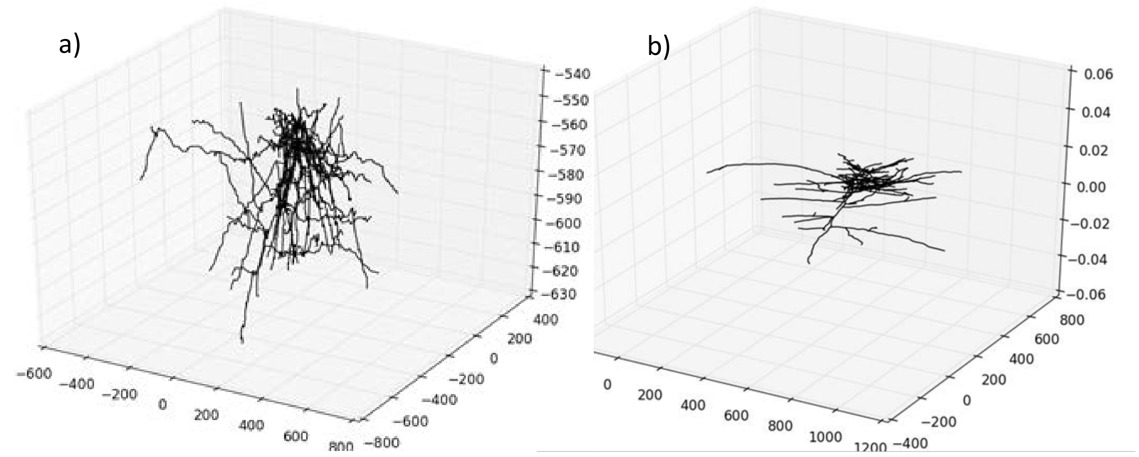


Figure 15. *Morphology of the rat's neocortex pyramidal cell extracted from NeuroMorpho database. a) 3D representation of the morphology imported from swc file and shown using brian2 library. b) The same morphology converted in 2D and shifted 500 µm right and up in 2D coordinates*

Figure 16, shows the difference between considering the neuron as soma, as in previous studies, and considering the neuron as morphology. As can be seen, there major differences between previous model and current model. First, in previous model, all the cells were in predefined locations and the neurons were considered connected if their distance was lower than a specific value. In this model, the locations are predefined, but depending on the shape of the neuron, some adjacent neuron might not be connected together. Second, in previous models, the distance that a neuron could reach was limited to its adjacent neurons (8 if the neuron is in the center of a square).

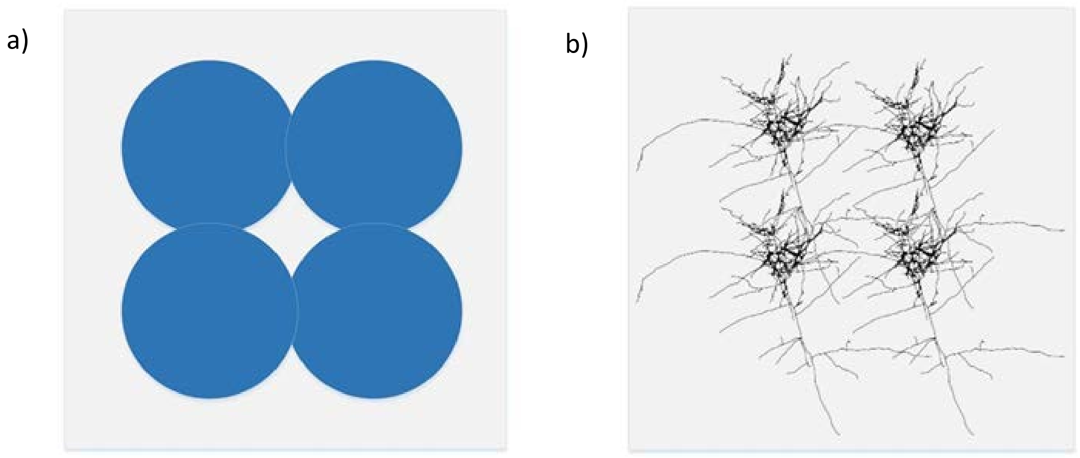


Figure 16. *Illustration of the difference between considering neuron as a) only a soma and b) a complete morphology in a simple MEA containing 4 neurons.*

In this model, however, a neuron might reach to 2nd level neighbors as well, depending on the length of its axons. Moreover, the strength of the connection in previous models was to be determined using some random parameters. In this model, the strength is estimated based on the intersection of axons of the reference neuron and dendrites of the target neuron. Although the initial connection of the neurons is only needed once in a simulation, yet the main obstacle of such approach is its heavy computation.

Each line of the *swc* file format indicates a single point in the structure of the neuron that create the whole neuron when connected. The pyramidal cell in Figure 14 has about 2200 axonal points and about 1800 dendritic points. If all of the permutations are taken into account, in a MEA plate with 1000 neurons a rough estimation give us a runtime of 500,000 years. This enormous runtime demands a novel algorithm for reducing the runtime to at least a few days.

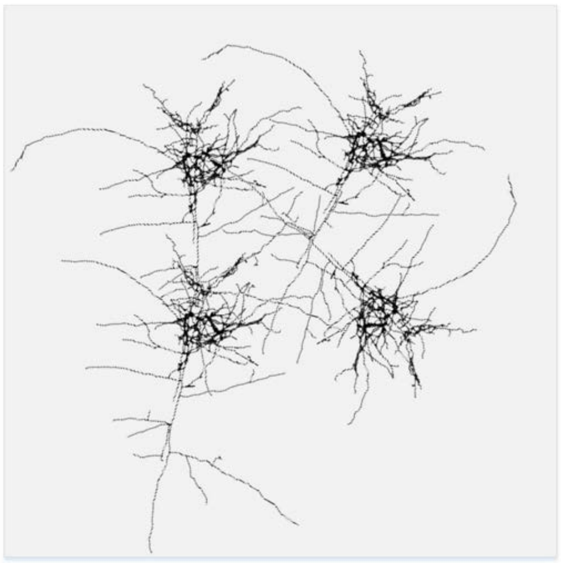


Figure 17. *Adding random rotation to each neuron for a more realistic connection between neurons*

This work propose an algorithm for reducing the runtime of this algorithm to 17 days. In the abovementioned simple approach, the calculation needed for calculating the intersections between axons and dendrites are performed between every two possible neurons. In other word, in a MEA plate with n neurons, n (n-1) sets of calculation must be performed. In the proposed algorithm, the accessibility level for each neuron is firstly defined by user. This level, ranging between 1 and $n\sqrt{2}$, defines the number of number of diametrical neurons that are to be taken into account. However, if a neuron is not connected to layer m, the algorithm would stop calculating for higher layers.

The algorithm then uses the building box of reference axons and target dendrites in a branch-wise order. Hence, if building box of an axonal branch is intersecting a building box of a dendritic branch, then the possible intersection of the lines constraining those branches are calculated and the intersection points are found. In the next step, these intersecting points between neurons are used to generate synapses that will eventually be used in brian2. The plotted result of this algorithm, applied on a network of 10 neurons (7 pyramidal cells and 3 basket cells) are depicted in Figure 18. As can be seen, the neurons are tightly connected together, forming a cluster-form ensemble.

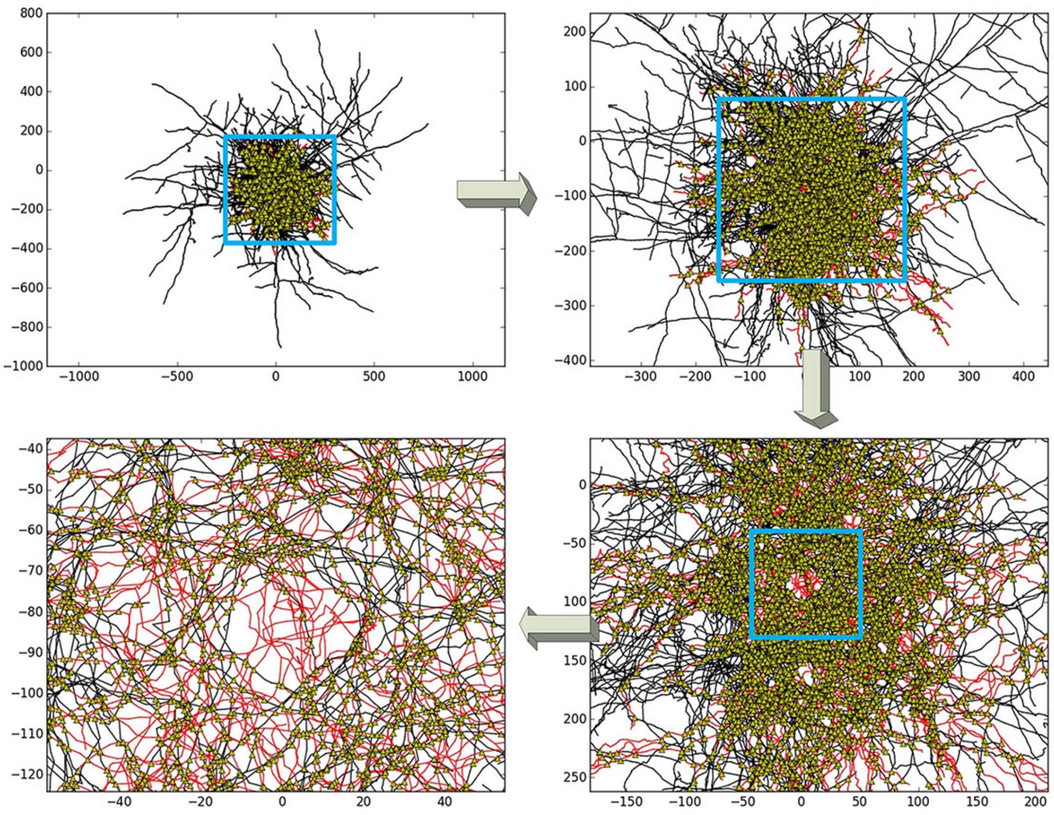


Figure 18. *The result of the connection_finder algorithm applied on a small network of 10 neurons with 7 pyramidal cells and 3 basket cells. Beginning from upper left figure, each plot is focused on part of the previous plot as indicated with a cyan square. In all four figures, the black and red lines represent axons and dendrites and the connection points are shown with green triangles.*

3.1.6 Mini-Compiler

In order for the whole system to work, it is demanded that the *Brian* is linked with the output of the connection maps resulted using the neuromorphology. In *Brian*, it is possible to define any number of synapse between two specific neurons of any group using a one line of code. In this definition of the synaptic connection it is possible to add other pairs of neurons to prevent redundancy in coding. However, this can be achieved only if the number of synaptic connection between all given pairs are the same.

For instance, it is possible to create 7 synaptic connection between neuron 2 and 6 with a line of code. The same exact line can be modified in a way that it creates 7 synaptic connections between neurons 3 and 4 or any other neuronal couple. However, as the number of synaptic connection changes to a number other than 7, it is not possible to create the synaptic connections with a single line of code anymore and another line is needed. Therefore, for a network consist of about 10 neuron in which each neuron could possibility connect to other 9 neurons, there might be 90 different connection, hence 90 lines of code. To address this issue, a mini-compiler is built that generates the required syntaxes for *brain* based on the output of the connections determined by morphologies.

3.1.7 Cell Stimulation

The stimulation capability of the MEA plate, i.e. the input of the system, can be handled by using creating an input and connecting it to a neuron group. There are two facts that can be considered for a more realistic representation of the stimulation. First, the monitoring over the cells activity can be removed from the results on the time that the cells are stimulated. Second, the effect of the input can be considered as a square but not with an equal current and voltage all over the area of the square, but in a way that neurons that are placed closer to the electrode are more affected than the farther neurons. This can be achieved by implementing the input based on a multivariate Gaussian distribution depicted in Figure 19.

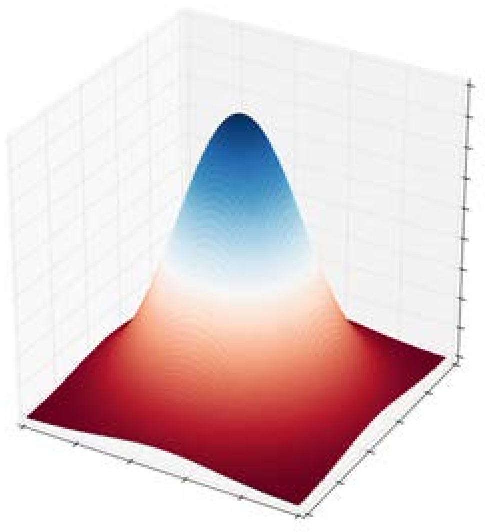


Figure 19. *Multivariate Gaussian distribution utilized for simulating the effect of a single electrode usable in both recording from neurons and stimulating them. The blue and red color represent maximum and minimum contact between the electrode and adjacent neuron, respectively.*

Figure 19 illustrate the affecting current in adjacency of each electrode of the model of MEA plate. In a standard MEA, there are 60 electrodes placed in an 8x8 layout grid. Remaining four electrodes are used as ground. The diameter of electrodes are either 10 µm or 30 µm with 100 µm or 200 µm inter-electrode distances, respectively [61].

Figure 20 illustrate the formation of the electrodes in the simulator. Note that some features in this figure is exaggerated for the matter of clarification. Nevertheless, the effect of the electrodes are completely customizable based on the parameters that are used for generating the multi-variant Gaussian distribution.

3.1.8 Linking the Components

Aforementioned components demand a right combination and practicable pattern of interaction. This interaction is analyzed according to the bottom-up approach presented in section 3.1. Starting from the intrinsic parts of the model, both sets of equations for neuron and plasticity model are inputted in the definition of the *NeuronGroup* in the simulation.

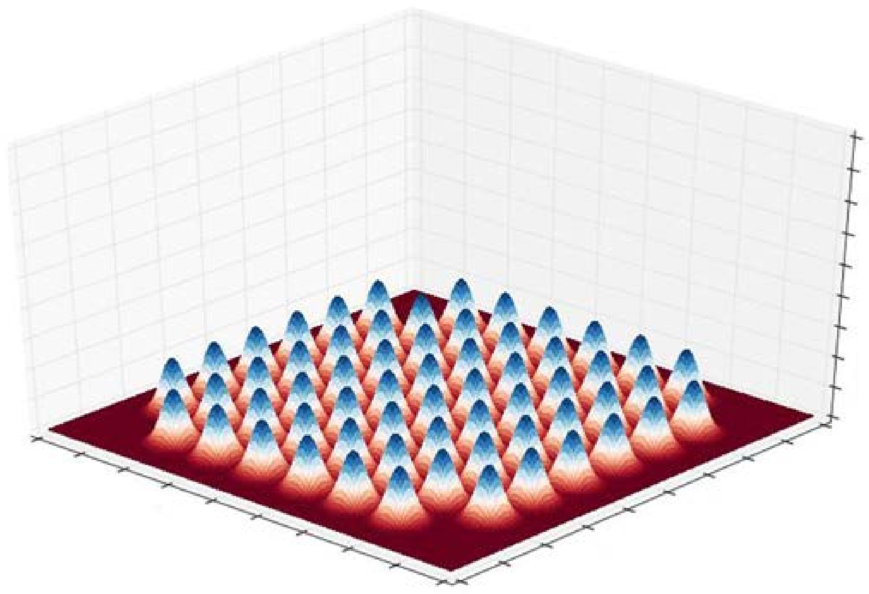


Figure 20. *Formation of the electrodes in a standard MEA simulated using multivariate Gaussian distribution. The corner electrodes are omitted since they are not used for recording and stimulating.*

The Stimulator as one of the major components of MEA, is inevitably linked with the simulator itself. As using the *state-monitors* provide us with the data for variety of parameters, by using either *SpikeGeneratorGroup* or *PoissonGroup* it is possible to stimulate the neurons. The former induce spikes based on the pre-defined times whilst the latter connects a Poisson input to the target group of neurons.

In the next step, each neuron of any *NeuronGroup* is assigned with a unique place in the MEA area. The location coordinates of a neuron is initially defined as an extra parameter for an instance of neuron. This allocation is done by choosing from the pool of available places resulted from the output of the distribution function.

The abovementioned components are used during initial stage of the simulation. On the contrary, the remaining components which represent two vital neuronal dynamics, i.e. cell death and connection, are dynamically taken into account during the simulation. In each internal step of the simulation, the conditions for both death and connections are reviewed. Some cells are then connected accordingly and some die based on the predefined timing.

3.2 Functional Connectivity Analysis

In this section, first a sequential algorithm for calculation of Cox coefficients and its confidence intervals are presented. Using this illustration the parts that can potentially be implemented in parallel are determined. In the next parts, two algorithm for implementing these parts on CUDA are proposed. Each algorithm dedicates the blocks of GPU to a unique part of the algorithm, resulting in a better performance of each in a specific types of spike train. In the final section of this part, method used for testing and evaluating these two algorithms is presented.

3.2.1 Component Analysis of Cox method

The algorithm for computation of Cox coefficients is depicted in the flowchart of Figure 21. As can be seen, the complex computation of first and second derivatives of the log likelihood function are implemented inside a triple nested for loop. It is clear from this representation that the first and second derivatives are independent, and thus, can be parallelized.

Regardless of this potential, a greater optimization can be achieved by optimizing the common data required for computation of both derivatives. Based on the expression of first and second derivatives, it can be understood that the great optimization can be happened by finding the values of influence function $Z_{B_i}(t)$, from all neurons and all their spiking times in advance.

The key approach for the two parallel algorithms is to use the blocks and corresponding threads of GPU to compute all the values of $Z_{B_i}(t_{lk})$ in parallel prior to calculation of cox coefficients. With a network of n+1 neurons, each with a recorded spike train with length of m spikes, $n.m^2$ values of $Z_{B_i}(t)$ must be calculated. As an example, a network of 1025 neurons with a recording time corresponding to a spike train with length of 1024 spikes will lead to computation of 536870912 values ($1024^3/2$) of influence function. Note that this amount of calculation is for computing the Cox method on only 1 neuron to find the effect of the other neurons of the network on this sole neuron. In the next two sections, the two proposed algorithms are presented. The illustration of these algorithms are based on the network structure of Figure 22.

In Figure 22, the cox coefficient from reference neurons, neuron 1 and 2, $\hat{\beta}$ is to be computed for the target neuron. Consider a simple scenario in which each neuron spikes three times. The goal is then to calculate the values of influence function Z() for each reference neuron at each spiking time of the target neuron. Although the amount of data used in this scenario is far below the required amount, this simple network topology and corresponding data clarifies the structure of the implementations of Cox method in GPU.

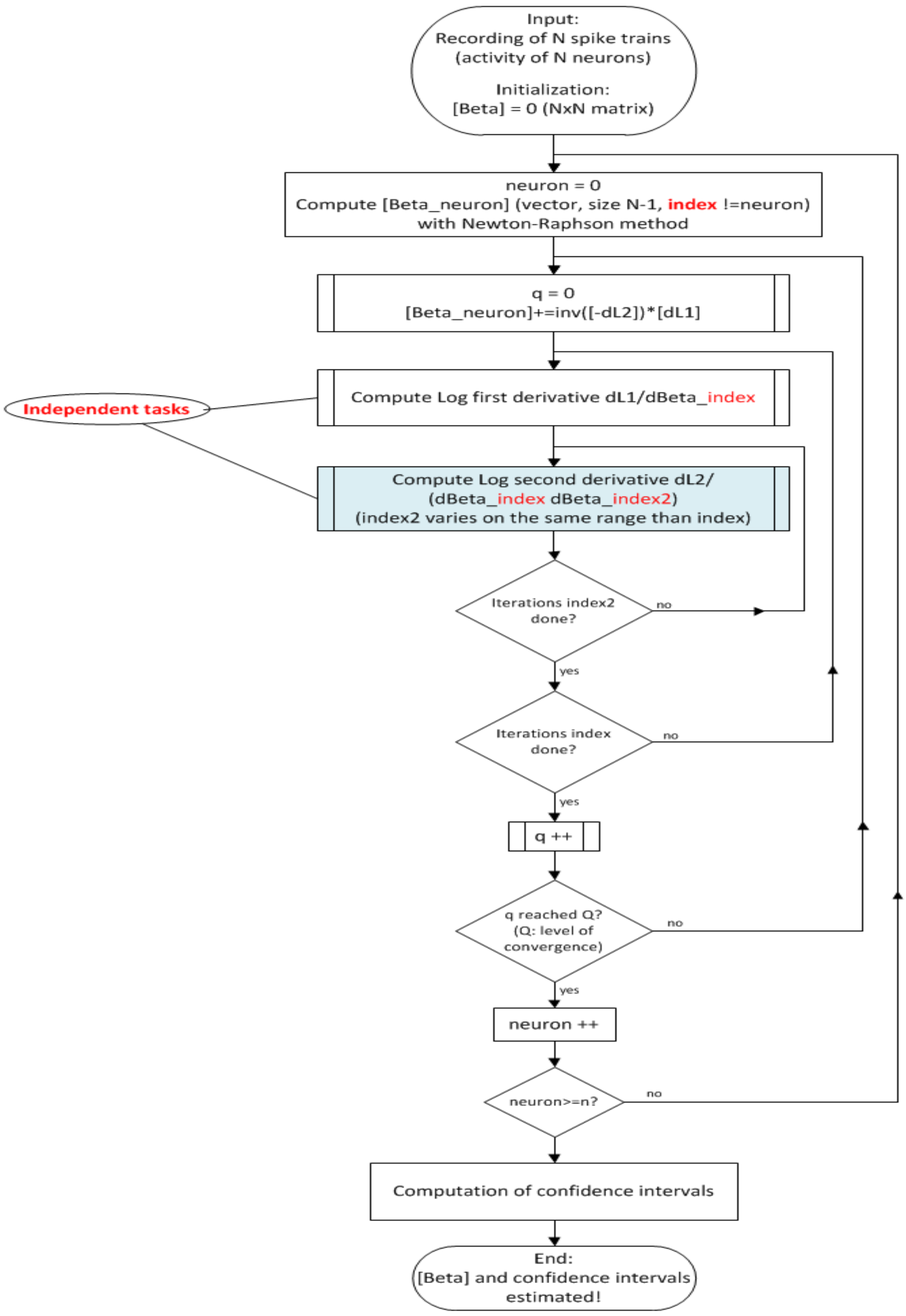

Input:
Recording of N spike trains
(activity of N neurons)
Initialization:
[Beta] = 0 (NxN matrix)
neuron = 0
Compute [Beta_neuron] (vector, size N-1, index !=neuron)
with Newton-Raphson method
q = 0
[Beta_neuron]+=inv([-dL2])*[dL1]
Independent tasks
Compute Log first derivative dL1/dBeta_index
Compute Log second derivative dL2/
(dBeta_index dBeta_index2)
(index2 varies on the same range than index)
Iterations index2
done?
no
yes
Iterations index
done?
no
yes
q ++
q reached Q?
(Q: level of
convergence)
no
yes
neuron ++
neuron>=n?
no
Computation of confidence intervals
End:
[Beta] and confidence intervals
estimated!

Figure 21. *Flow chart of the main tasks of the Cox method presented sequentially. The notations dL1 and dL2 in this chart correspond to the first and second differential of the log likelihood L according to the coefficients of connectivity, $\vec{\beta}$. Thus, dL1 corresponds to the gradient of the log likelihood and dL2 corresponds to the Jacobian of this gradient, also called the Hessian matrix of the log likelihood.*

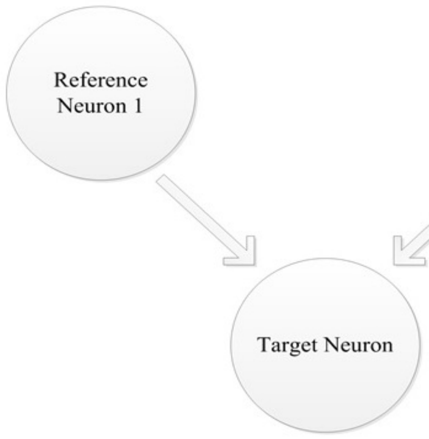


Figure 22. *Simple network topology considered as example for describing the two GPU accelerated implementations of the Cox method.*

3.2.2 1st algorithm

In this algorithm the Z values of all reference neurons for a specific target neuron's spike is calculated in a single GPU block. The timing of this specific spike is resulted from sorting as depicted in Figure 6. Each of this influence function values is calculated using a thread of that block, as in Figure 23. Since each block calculates the values of Z for a unique time $t_{x_{(i)}x_{(j)}}$ and for all reference neurons the blocks of this algorithm will experience an increase when the number of neurons grows. For this reason, the performance will not face a great deterioration y increasing the number of neurons. Using this implementation in a network with *n+1* neurons whilst target neuron has m Inter Spike Intervals (ISI), each block will always have *n* threads. The grid in this algorithm contains m*m blocks.

3.2.3 2nd algorithm

In this approach, the row and column of grid are dedicated to the Z values of specific reference neuron and specific target time respectively. Hence, the threads within a block always contain Z values of the same reference neuron (e.g. *ref1*) and have the same value for the first index of t_{ij} (e.g. t_{3j}). All Z values for all the smaller ISIs of that specific time i.e. $Z_{ref_n}(t_{ij})$ with $j \leq i$, are computed using the threads of a block.

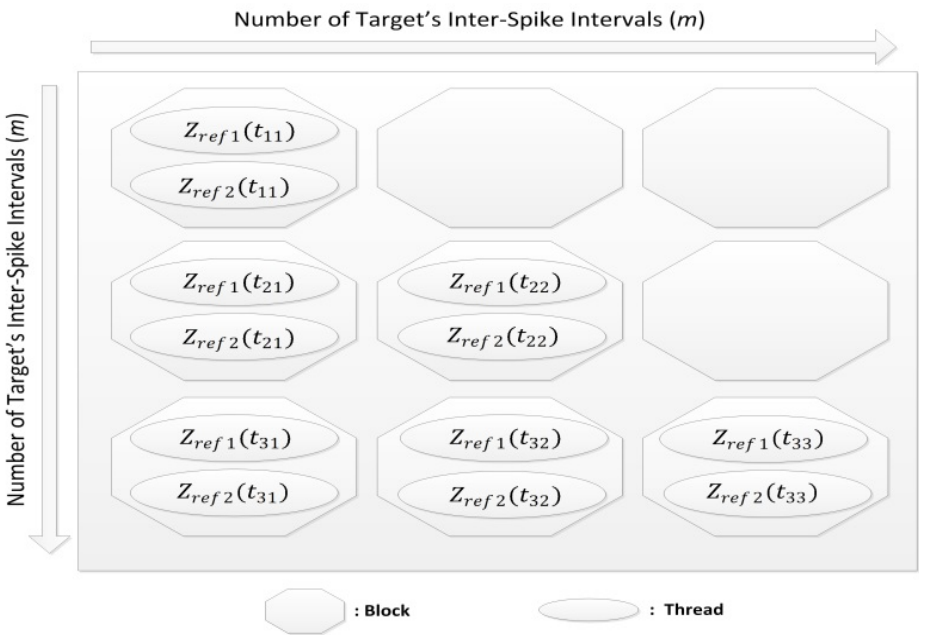


Figure 23. *Schematic of the grid formation for the first algorithm where $Z_{ref\ n}(t_{ij})$ indicates the Z value of reference neuron n at time t_{ij}.*

As for the size of the grid and blocks in this algorithm, the former has a size of n*m blocks and the latter contain m threads, with n and m indicating the same metric as previous section. The growth order of grid size is linear with reference to ISIs and number of neurons. The schematic of this algorithm is presented in Figure 24.

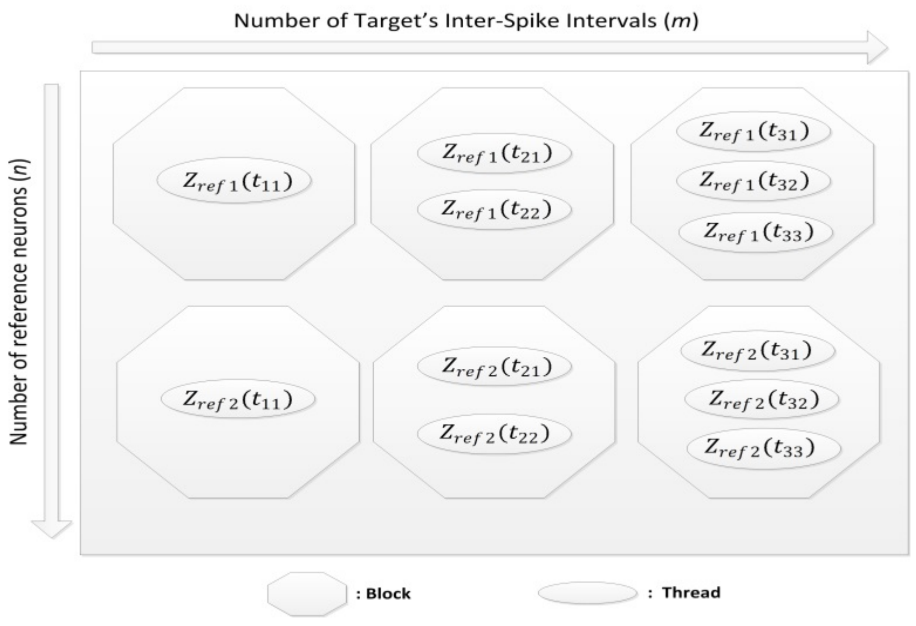


Figure 24. *Schematic of the grid formation for the first algorithm where $Z_{ref\ n}(t_{ij})$ indicates the Z value of reference neuron n at time t_{ij}.*

3.2.4 Test and performance evaluation of algorithms

The algorithms were compared in two steps. Firstly, both algorithms are compared with sequential implementation in CPU. In this comparison the performance of CPU is evaluated and compared to that of GPU with two variables of interest as the number of neurons and average number of spike. Secondly, same comparison criteria are applied to compare the performance of GPU algorithms with a more massive data set. For the first step, the number of neurons varies from 3 to 18 and the number of spikes for tests is between 30 and 165. As for the second step, the average number of neurons varies between 3 and 70 and the number of spikes varies between 160 and 945. Note that the results of both GPU algorithms in comparison with CPU were examined and with the same dataset, all three implementations produced identical results. It worth mentioning that in a network of *n* neurons, acquiring full network connectivity demands the method to be executed n times. For this reason, all runtimes provided in this work, consider the duration needed for attaining the full connectivity of the network.

4. RESULTS AND EVALUATION

In this section final result of both parts of this work are presented. Note that the partial result of each component and sub-component of the section 4.1, was presented in different parts of section 3 and in this section, the output of the framework as a system is presented.

4.1 MEA simulator

The simulation presented in this section is based on a small model of MEA plate containing 1000 neurons over an area of 7,480,000 μm^2. Among the 1000 neurons plated in this simulated MEA, 493 neurons survived after applying the death rate. There are 70% percent excitatory pyramidal cells and 30% inhibitory basket cells, each from 5 different types (in total 10 types of neurons). The type and number of neurons in the simulated MEA is illustrated in Table 2.

Table 2. *Distribution and types of the total of 493 neurons in the simulated MEA.*

	Excitatory (Pyramidal) Cell Types					Inhibitory (Basket) Cell Types				
Neuron Type	D27B	D20D	I03481	C300898C-P2	C031097B-P3	C070600A4	C010600C1	C010600A2	BE23B	BE49B
Number of Neurons	69	70	70	69	69	29	29	29	30	29

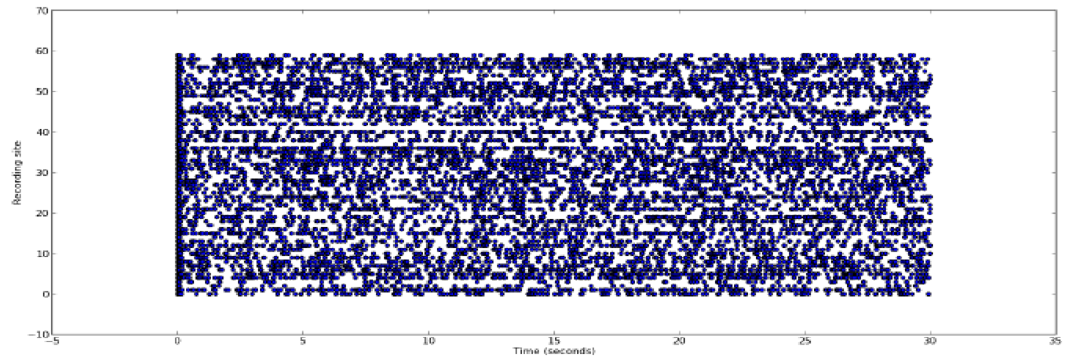


Figure 25. *Simulation of the neuronal behavior in MEA plate previously presented in [57]*

The result of the simulations in [57], as presented in Figure 25, suffers from over-activity in neurons. These over-activity is smoothened in current simulator, presented both in Figure 26 and 27.

Figure 26 presents the final result presenting the neural activity (spiking) of all survived excitatory neurons. As can be seen, the vertical lines indicating the synchronous activity between neurons are not observable in Figure 25 [57]. In Figure 26, however, indications of synchronous activity between neurons are observable. In Figure 27, the simulation result based on the recording sides is presented.

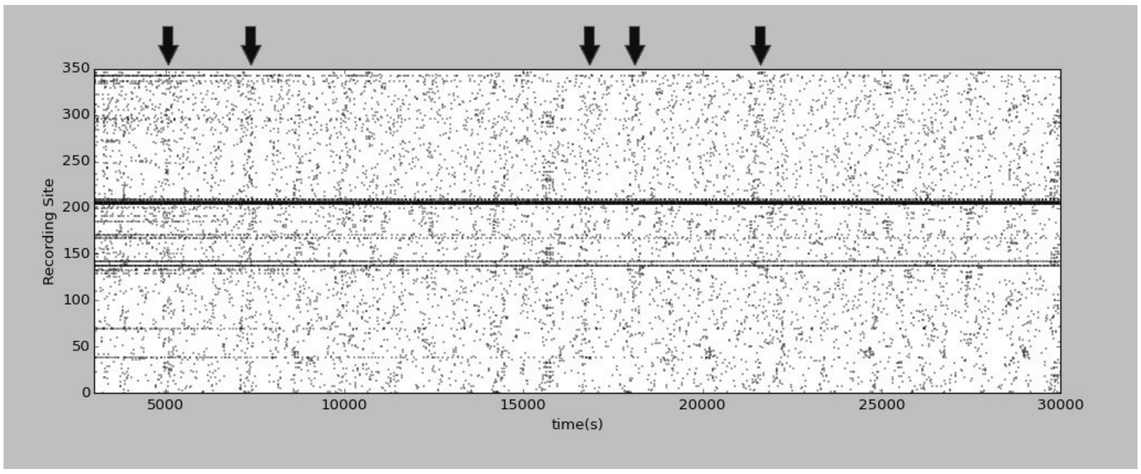


Figure 26. *Output of the current simulator for all survived excitatory neurons in a simulated network of 1000 neurons. Black arrows shows indications of synchronous activity between neurons.*

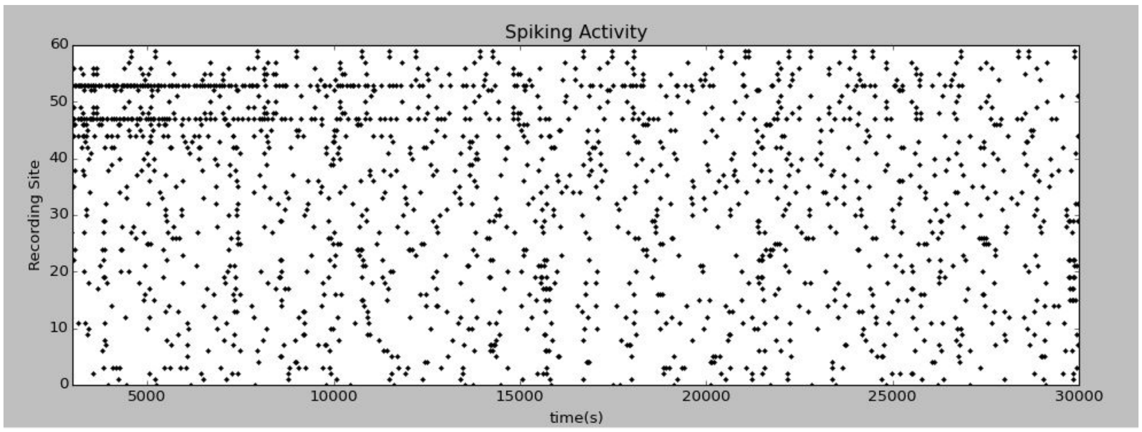


Figure 27. *Output of the current simulator for survived excitatory neurons near the electrodes location in a simulated network of 1000 neurons*

4.2 Functional Connectivity Analysis

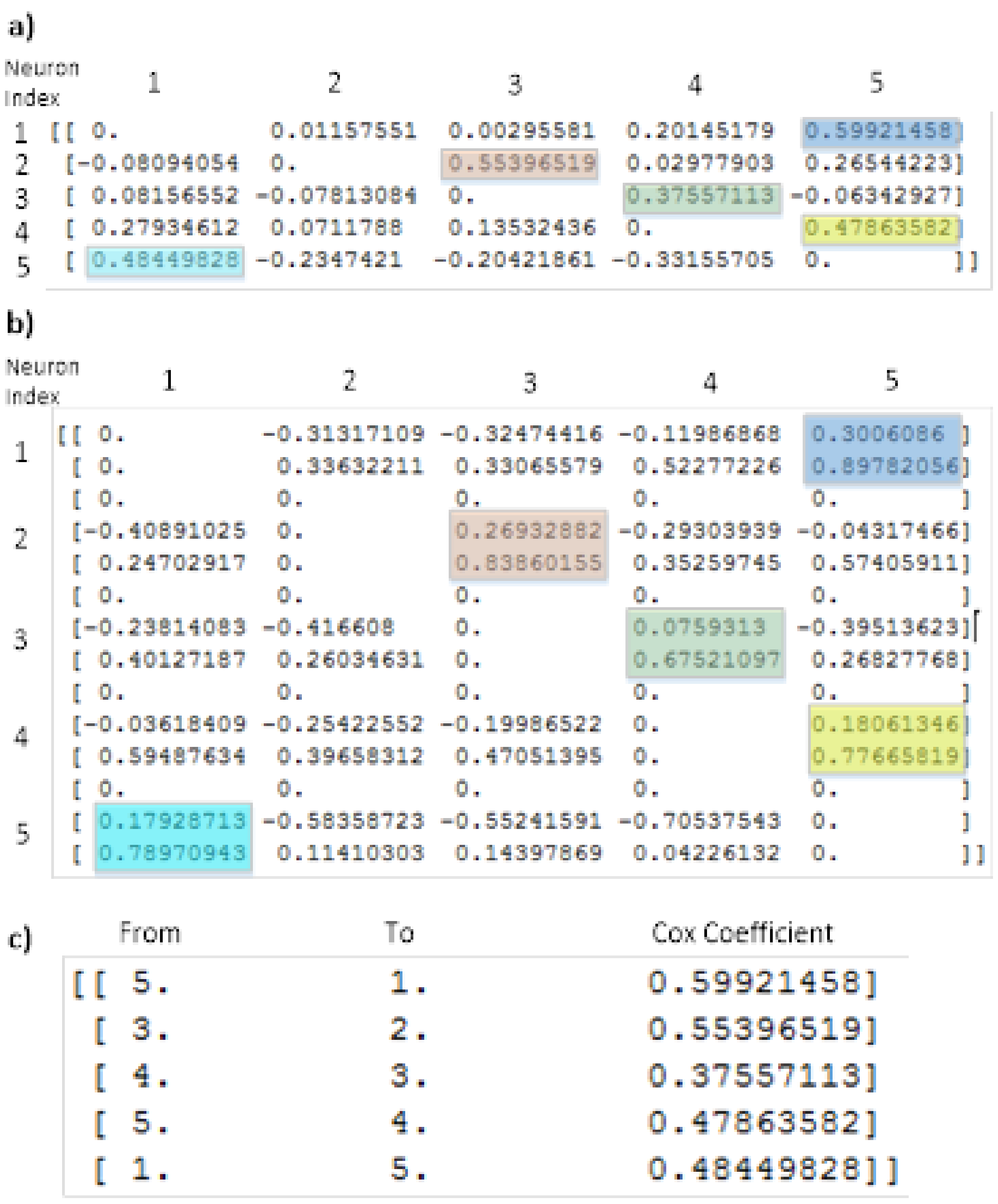
Figure 27 shows the output result of the cox method applied over a small network consist of 5 neurons. The output consists of three main parts: adjacency matrix of confidence interval for beta values (Figure 14-a), adjacency matrix of confidence intervals of the beta values of the previous part (Figure 14-b) and the output of the connectivity map (Figure 14-c).

As mentioned in section 2.3, Cox method is statistical in nature. Hence, it is essential to consider the confidence interval for each beta (cox coefficient) value. As can be seen in Figure 27-a, for every neuron pair there is a beta value and it cannot be determined if that value indicates a real connection or not. For confirming the integrity of the beta value, the corresponding confidence interval should be checked. In case the confidence interval does not include zero, the connection between the pair can be taken as a real connection and the strength of the connection can be considered as the beta value. In Figure 27, values with confidence intervals that does not contain zeros are shown in colors. Finally based on these values, the final table is formed as Figure 27-c.

The results of this sections is presented in two sub-sections: first, the comparison between performance of CPU and that of both GPU algorithms are made. By justifying the uselessness of CPU for such statistical method, the performance of algorithms are evaluated against each other. In the second part larger datasets are used and comparison is performed based on two major parameters of activity recording: network size, and duration of recording (i.e. length of spike trains).

4.2.1 CPU vs GPU

Figure 28 and Figure 29 illustrate the comparison between the performance of two GPU algorithms and CPU. In Figure 28, the comparison is made based on the number of neurons in the network. In this figure, the results of the comparison are provided with regard to networks containing 3 to 18 neurons with a fixed recording time of 5s (this duration is equal to about 70 spikes per neuron). This comparison shows an exponential growth in runtime of CPU with increase of network size. The GPU runtime, however, stays under 25 seconds with a very slight growth. In the largest network containing 18 neurons, CPU takes more than 570 seconds whilst both GPU algorithms take less than 25 seconds.



a)

```
1 [[ 0.          0.01157551  0.00295581  0.20145179  0.59921458]
2  [-0.08094054  0.          0.55396519  0.02977903  0.26544223]
3  [ 0.08156552 -0.07813084  0.          0.37557113 -0.06342927]
4  [ 0.27934612  0.0711788   0.13532436  0.          0.47863582]
5  [ 0.48449828 -0.2347421  -0.20421861 -0.33155705  0.        ]]
```

b)

```
1  [[ 0.         -0.31317109 -0.32474416 -0.11986868  0.3006086 ]
    [ 0.          0.33632211  0.33065579  0.52277226  0.89782056]
    [ 0.          0.          0.          0.          0.        ]
2   [-0.40891025  0.          0.26932882 -0.29303939 -0.04317466]
    [ 0.24702917  0.          0.83860155  0.35259745  0.57405911]
    [ 0.          0.          0.          0.          0.        ]
3   [-0.23814083 -0.416608    0.          0.0759313  -0.39513623]
    [ 0.40127187  0.26034631  0.          0.67521097  0.26827768]
    [ 0.          0.          0.          0.          0.        ]
4   [-0.03618409 -0.25422552 -0.19986522  0.          0.18061346]
    [ 0.59487634  0.39658312  0.47051395  0.          0.77665819]
    [ 0.          0.          0.          0.          0.        ]
5   [ 0.17928713 -0.58358723 -0.55241591 -0.70537543  0.        ]
    [ 0.78970943  0.11410303  0.14397869  0.04226132  0.        ]]
```

c)

```
   From          To          Cox Coefficient
[[ 5.            1.          0.59921458]
 [ 3.            2.          0.55396519]
 [ 4.            3.          0.37557113]
 [ 5.            4.          0.47863582]
 [ 1.            5.          0.48449828]]
```

Figure 28. *Output of Cox method in this study. a) Adjacency matrix of Cox coefficients b) Adjacency matrix of confidence intervals for Cox coefficients in a. The Cox coefficients are considered as a real connection if their corresponding confidence intervals does not contain zero. In such a situation, the value of confidence interval is taken as the connection strength. c) Adjacency matrix of connectivity result of the network. The values strength of each connection is extracted from the cox coefficient adjacency matrix.*

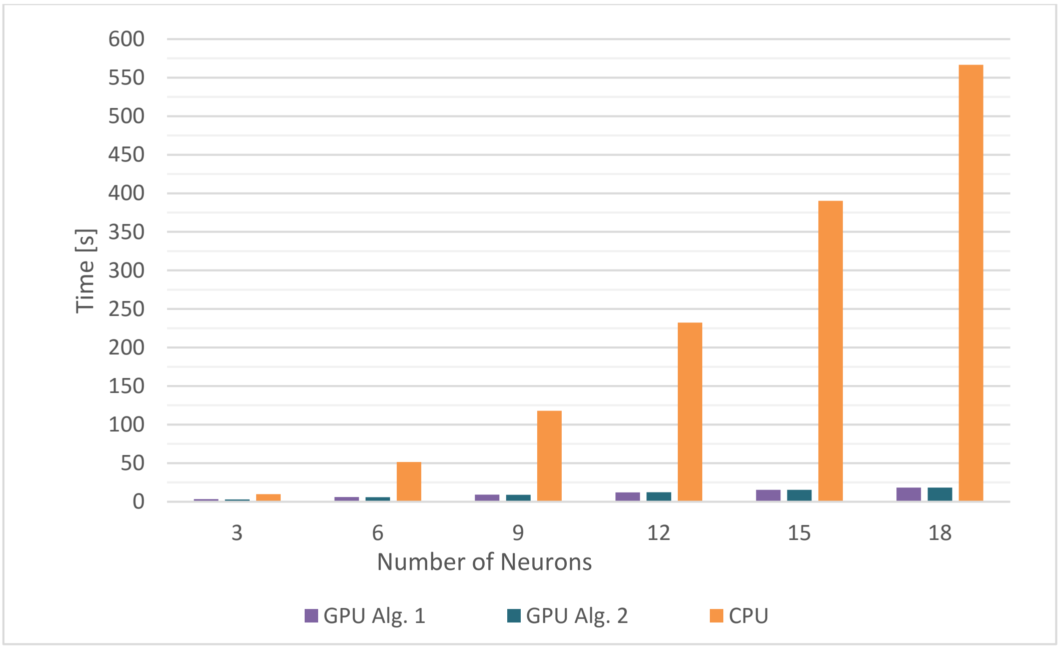


Figure 29. *Performance (runtime) comparison of the CPU implementation of the Cox method with two GPU algorithms with number of neurons in the network as the variable.*

The comparison in Figure 29 is based on the average number of spikes recorded for each neuron. The length of spike trains for a network of 5 neurons varied between 33 and 165 spikes in average which corresponds to recording time of 2s to 10s, respectively. The same exponential growth in CPU runtime with increasing the length of spike trains is observable in Figure 29 again. The runtime of CPU for spike trains of length 165 average spikes takes almost 155 seconds whilst both GPU algorithms remains under 10 seconds.

Both of abovementioned evaluations were maintained over small networks with short length of recording. Increasing either number of neurons in the network or length of recording will result in an overshoot in runtime of CPU which makes the comparison of GPU algorithms impossible owing to the fact that their timing bars will shrink to an unclear value near zero. The next section covers the performance evaluation of the GPU algorithms on larger datasets.

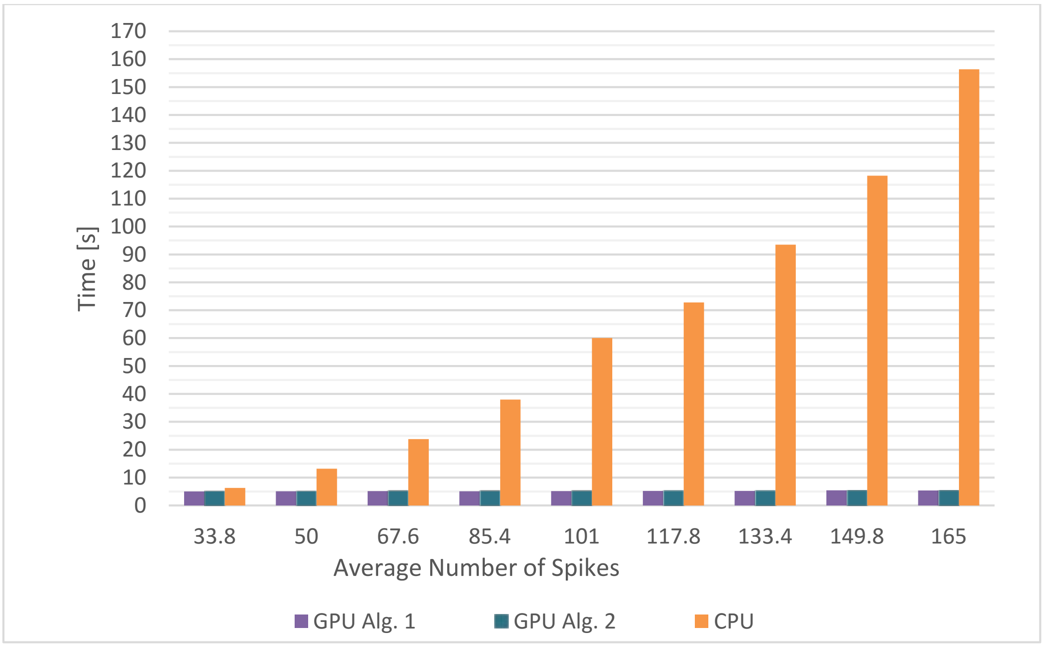


Figure 30. *Performance (runtime) comparison of the CPU implementation of the Cox method with two GPU algorithms with length of spike train in the network as the variable.*

4.2.2 GPU alg. 1 vs GPU alg. 2

Similar to previous section, the performance comparison of the two GPU algorithms is maintained based on network size and length of recording. The used datasets, however, vary in size enormously. In addition, this section focuses mostly on the runtime of the parallelized parts of the cox method, i.e. the computation of the values of the influence function Z(), as the remainder has to be calculated on CPU sequentially. For this reason, the next two plots illustrate GPU execution time only for computation of all necessary Z values which will consequently be used in sequential part of program.

First, the algorithms are compared based on different sizes of network, ranging from 5 to 120. In this evaluation, the average length of each spike train is 10s equal to average of 150 recorded spikes per neuron. As can be observed in Figure 30, with smaller networks, first GPU algorithm has a better performance compared to the second one (for networks up to 32 neurons). However, the growth of runtime of first algorithm with 25ms for a network of 5 neurons is much smaller than that of second algorithm.

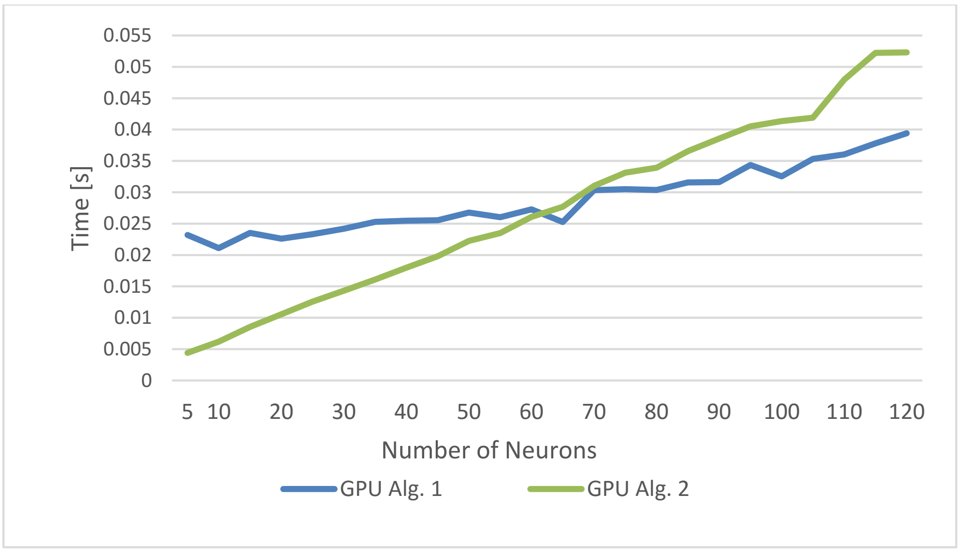


Figure 31. *Comparison of execution times of between the 2 GPU implementations of the Cox method, as a variable of the number of neurons in the studied network.*

In the second evaluation, the algorithms are compared based on the data generated for a very small network of 5 neurons. In this evaluation, the duration of recordings varies from 10 second to 1 minute, equal to average spike trains ranging between of 157 to 945 spikes. In this evaluation, since GPU grid size grows with number of spikes, first algorithm runs out of memory with grids of size bigger than 760x760 blocks. The runtime in this algorithm, increases from less than 0.5s for spike trains with length smaller than 490 spikes, to almost 2s at 760 spikes. For algorithm 2, however, the increase in runtime is much slower and is still under 1s for recording with length of ~945 spikes.

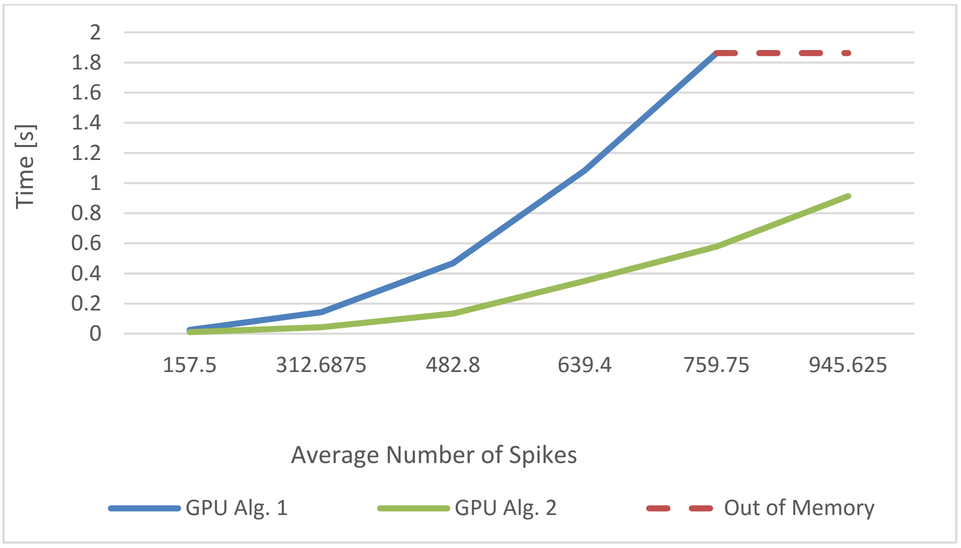


Figure 32. *Comparison of execution times of between the 2 GPU implementations of the Cox method, as a variable of the average number of recorded spikes in the studied network.*

5. DISCUSSION

5.1 MEA Simulator

In this section, the discussion regarding to both parts of this study, i.e. realistic modeling of the MEA plate and connectivity analysis based on the recorded data, are provided. First, the presented model is compared with four other models in the literature previously provided by Shultz [57], Chao et al [62], Bruzzone et al. [63] and Lenk & Priwitzer [64]. Each of these models are discussed in following sub-sections. This is followed by a thorough discussion on the implementation of the Cox method.

The essence of MEA functionality is the connection between the cultured neurons. In this regard, three different actions was previously considered in the model proposed by Shultz. Firstly, the growth rate and direction of the axons of a neuron, secondly, connecting of the adjacent neurons and thirdly, the migration of the neurons. For the first two behaviors, some limitations was considered too, owing to the fact that there are limitations to formation of new connections in a real MEA plate. Such limitations are resulted from chemical interactions between cells. In the stochastic model used in [57], which was previously presented by Kahng et al [65], the movement of the growing end of the axon happens in a form of a random walk on a grid. The direction of this growth is considered to be in 8 directions in Shultz’s model: up, down, left, right or the diagonals between them. After each step two neurons are logged as connected if one of the walking end is with 20 μm of the other. Moreover, the connection are unlikely to happen with the reference neuron itself and is more likely to happen with its neighbors [66].

Gafarov in [67] indicated that whilst spiking, developing neurons attract other axons towards a cell. Similarly in this model, the intense activity of the neurons resulted from stimulation in results in customized development in specific parts of the simulated MEA plate. Though, Shultz puts some limitations to this development based on two studies: first, the paper proposed by Segev et al [68] showing that the cultured neurons will grow and create new connections to at most 10 neurons. Second, the model presented by Patel et al [69] in which the out-degree is reproduced using a Poisson distribution with a mean of 22. It uses a Gaussian distribution to determine the connection probability of adjacent neurons with a straight-line distance between them. This parameters get the maximum value of connectivity around 200μm from the soma. Inducing connection between the neurons by stimulating the neurons can be used for training the network for performing variety of function, as the functionality of the neuronal network pertains for its internal connectivity structure.

The model provided by Abraham Shultz, has some of the components taken into account in this study into account. Starting with model of the neuron, although that model uses the Izhikevich model of both excitatory and inhibitory neurons with considering “Fast

Spiking (FS)” behavior in the inhibitory neurons. Additionally, Shultz’s model utilizes the midpoint displacement fractal algorithm for plating the neurons in the simulated MEA. Moreover, it takes the death rate into account and removes the 40-60% of the cells in the beginning of the simulation.

Shultz’s model, implement the cell connection and growth based on a completely different approach. It uses a Gaussian distribution to determine the connection probability of adjacent neurons with a straight-line distance between them. This parameters get the maximum value of connectivity around 200μm from the soma.

In that model, it is mentioned that it is possible for each cell to connect to any other node in the MEA plate. The positive point of that model is applying some limiters for the connection formations, extracted from literature. As mentioned in section 3.1.5, due to numerous branches in each cell and small size of a MEA, many synaptic connections might happen between two neurons, over-strengthening the connection in the whole MEA. The model provided by Shultz, has a realistic model of electrodes for recording from the neurons, which is also implemented in current study both for recording and stimulating.

The model proposed by Chao et al, implements 1000 LIF neurons with total of 50,000 synapse. The 1000 LIF neurons in that model are randomly distributed in a virtual MEA with size of 3mm by 3mm, i.e. area of 9,000,000 μm^2. In this model, all the synapses were frequently dependent in order to model the synaptic depression. Moreover, 70% of the neurons were excitatory implemented with STDP, and the electrode formation was similar to that of a standard electrode, i.e. 8 by 8 grid of electrodes, 60 of which are used for recording and stimulating.

The model provided by Bruzzone et al. is used to be connected to biological neural network. In this model, they used a network of 100 Izhikevich neurons with 80 excitatory and 20 inhibitory randomly placed on the area of the MEA. The excitatory population of the neurons comprised regular spiking, intrinsically bursting and chattering neurons. Similar to Shultz’s model, the model provided by Bruzzone et al. also considers the Fast Spiking neuron for inhibitory neurons. The connection between the neurons are also determined based on the random distribution with average degree of 75 and uniform distribution was also utilized for setting the synaptic weights. This uniform distribution was specified separately for excitatory and inhibitory populations. The model of Bruzzone, was implemented over the NEST simulator.

The goal of the model presented by Lenk and Priwitzer [64] is to simulate the concentration-response curves that were previously observed in in-vitro experiment. The cells in this model, which are described as black boxes, have two states of ON and OFF and each neuron can have multiple inputs whilst it only produce a single output. To determine the occurrence of spikes, Poisson process was employed. Each cell can be connected with either inhibitory or excitatory synapse. The weight of the synaptic connection could vary. Apart from these, the network is fully connected featuring a spike time history. The final simulation in this model ran for 10 seconds with a network of 100 neurons, containing 90 inhibitory neurons and 10 excitatory neurons.

5.2 Connectivity Analysis

The sequential implementation of the Cox method was maintained and ran on MATLAB and in next step, PyCuda was used for implementing two parallel algorithms. The experimental platform was a laptop equipped with an Intel Core i7-4702MQ CPU and NVIDIA GK208M (GeForce 740M GT) with 2 GB video memory. The results of all three implementations (CPU and two algorithms on GPU) were checked and compared with another and their integrity were confirmed. With identical datasets all three algorithms produced identical output.

Note that each instance of running Cox method as described in this study, solely take the computation of connectivity coefficient to one neuron of the network into account. For this reason, for calculating the full connectivity map of a network with N neurons, it needs to be ran N times. Therefore, all the runtimes provided as a result in this work, considers the duration needed for calculating the full connectivity of the network.

The exponential growth in CPU runtime, demand a long time for providing the connectivity map of a network with a minimum realistic size of 60 neurons (one neuron per electrode). With the small dataset, CUDA implementations provide results at almost constant time (under 25s for 18 neurons, and under 10s for an average of 156 spikes).

Applying the sequential implementation over a larger dataset will definitely rule out the use of sequential version of Cox method. For example, a network composed of 5 neurons with an average recorded spikes of 950 per neuron demands almost 17 minutes for computing the Z values for a target neuron on CPU, taking almost ~90 minutes for calculating the connectivity map of the entire network. GPU algorithms, however, needs ~0.91 second for each Z matrix and 2 minutes for creating the whole connectivity map. Similarly, a network containing 70 neurons with a recorded data of average 80 spike per neuron demands ~3mins for computing the Z values of a single neurons. Consequently 1.5 hours are needed to compute the entire connectivity. The first GPU algorithm, however, needs 7ms for each Z matrix and ~1.5 minutes for total connectivity. As mentioned, the performance of the first algorithm is better when the number of neurons per network is increased. Conversely, the second algorithm performs better when length of spike trains are increased.

The performance of the two proposed algorithms varies based on the characteristic of the data. Algorithm 1 supports connectivity analysis of datasets with larger number of neurons. This algorithm, however, is weak against long spike trains since the increase of spike train length, induces an increase of GPU grid size. On the contrary, second GPU algorithm, has better performance for long spike trains compared to the first one. In turn, this algorithm outperformed when the number of neurons increases, provided that the extensive number of neurons does not cause the algorithm to go out of memory. This analysis, gives us an estimation of the possibility to feed the first algorithm with a real dataset of up to 1024 neurons and in the second algorithm with maximum of 1024 spikes per train. Respectively, there is a limit for number of number of spikes in each train and total number of neurons in the network which is fully dependent on the memory of the GPU.

Selecting the proper algorithm is inextricably linked to the number of neurons and the time of recording of the dataset. Theoretically, Cox method provides accurate results with recordings of length 512 spikes equal to approximately 40 seconds of recording [24]. This limit provides an important selection point between algorithms. Recordings need to last at least 40s. Obviously, in case of longer trains, they can always be split into recording parts of 40s. Now assuming a record length of 40s, i.e. equal to 512 spike per neuron, first algorithm should be selected. Conversely, when the connectivity map of a network with number of neurons fewer than 512, second algorithm should be chosen.

6. FUTURE WORKS

6.1 MEA simulator

In this section the possible future developments with the goal of extending the MEA simulator are presented. The goal of these development are twofold: performance improvement and making the model more realistic.

The first step to find the solution for performance improvement of the model is to find its performance bottlenecks. Two main performance bottlenecks of the model are *brian* simulator and the stage at which the initial connection are found based on the morphologies. These two bottlenecks are also directly linked together. Since using the morphology for finding the initial connections might lead to generating many instances of synapse objects, it will eventually reduce the performance of *brian*. The best solution for improving the performance of *brian* simulator for this work, is to employ the multithreading with OpenMP, for which instructions are available in brian documentation.

As for the latter performance bottleneck, although the current algorithm is only needed to be run once, it still demands a performance improvement due to long runtime. By reviewing the utilized algorithm, it will be understood that the nature of the algorithm is parallelizable. The simplest parallelizing solution, would be to implement the algorithm over GPU and distribute the calculation of each cell, which are not dependent to each other, to separate cores. Another solution would be utilizing the Hadoop for a map-reduce implementation of the algorithm.

Regarding the behavioral aspect of neurons, the growth of neurons is an important behavioral aspect that could be improved in this model. The main drawback of the utilizing the morphologies is that each cell is already mature, in matter of size, from the beginning of the simulation. In reality however, the axons and dendrites grow over the MEA and make the connection. Another in-vitro behavior of the neurons is the migration during the early DIV [68]. These migrations will lead to formation of the clusters. The intra-cluster connectivity of the neurons is very dense whilst the inter-cluster communications happens to be sparse. The implementation of neuronal migration, even if it is few in matter of distance, is a behavior that makes the model more realistic.

A general improvement in this model can be achieved by implementing different types of neuronal behavior, i.e. Fast Spiking, Chattering, Intrinsically bursting, etc., using the known parameters of Izhikevich equations. In that case, the simulator can generate the proper neuronal behavior based on the morphology.

Eventually, a decent model of MEA plate can be used in variety of applications, ranging from bio-integrated system to researches in different areas of neuroscience. Other applications of a realistic simulation of MEA plate would be to evaluate a model before implementing it using a biological neuronal network. For instance, a bio-integrated wireless sensor network, capable of detecting different shapes, e.g. cross, circle, rectangle, triangle, etc., can be simulated and tested before spending budget on the expensive lab facilities.

For the second part of this study, i.e. connectivity analysis of the neuronal network, an important improvement would be enhance the improvement to make it possible to be ran on CUDA clusters. Moreover, there are some memory management techniques available for CUDA that would eventually help both algorithms to run on larger number of neurons as well as longer spike trains.

The second part of this study is also advantageous with regard to intervals during which the connectivity map is going to be extracted, making functional connectivity evolution analysis available. By running the cox method on shorter periods, it would be possible to calculate $\frac{d\beta}{dt}$ to predict the succeeding Beta values, facilitating a more precise control over stimulations for obtaining a pre-defined topology. In this regard, controlling the artificial changes in the network structure could enable its engineering for specific cognitive tasks.

7. CONCLUSION

7.1 Simulation of neuronal activity in MEA plate

The main contribution of this study is the development of a framework for simulating the behavior of the neural network in MEA plate. This was achieved by considering and implementing seven components in a real MEA plate, namely neuron, synaptic plasticity, electrodes, neuromorphology, cell distribution, cell connections and cell death.

Whist the model of the neuron and its synaptic plasticity in the plate is implemented as that of Izhikevich and STDP, all remainder components are configurable: the electrode arrangement can be changed based on the type of the MEA and the range of the electrodes can be configured based on the type of electrodes. Additionally, this framework can be inputted with neuromorphology file extensions, i.e. *swc*, download from the large database of the neuromorphology entitled NeuroMorpho. The distribution of the cells in this system is determined using the nested inversed diamond-square algorithm and the connection between cells are determined using their real morphology.

Aforementioned components can be customized using numerous parameters. The most important parameters are the size of MEA and somas in µm, number of electrodes, total number of neurons. Moreover, MEA can be inputted with different types of neurons with a provided distributions. For instance one might want to simulate a MEA with 30% inhibitory basket cells, 30% self-firing neurons and 40% excitatory pyramidal cells. Most of the other parameters, although configurable, but would be determined by the framework automatically. Model of the cell and corresponding plasticity can also be configured and be fed to *brian* simulator.

The provided framework and MEA model is advantageous in matter of employing state-of-the-art Izhikevich model combined with STDP as well as the plating algorithm which provides a realistic distribution of the neurons in the MEA. Additionally finding the connections based on the morphology of each cell in the MEA plate is an accurate approach. However, finding connections for a set of 1000 cells still takes a rather long time (about 2 weeks). Though, the initial connections has only to be determined once and simulation of that specific MEA can be started based on the saved connection map.

Definition of the synapses in *Brian* simulator can be troublesome in case the number of synaptic connection between neuron couples varies. In other words, due to different number of synaptic connection between neuron couples, each needs to be defined in a separate line. For instance, with a very small network of 10 neurons in which each neuron has connection with other 9 neurons, 90 lines of code would be required to define the synapses. This framework provides a mini-compiler which compiles the output of connectivity map inside *Brian*.

7.2 Connectivity analysis of neuronal network

In the second part of this work, Cox method, as a robust method for connectivity analysis of neuronal network based on spike train data, was implemented on GPU and its performance was evaluated and compare with CPU. The GPU implementation was maintained based on two different algorithm each proper for a specific circumstance. Both GPU algorithms focus on accelerating the calculation of the values of the influence function Z() which are eventually saved in the form of a 3D matrix. Apart from that, the calculation of the Hessian of ML was also accelerated with GPU.

The performance of these algorithms were evaluated based on increasing number of neurons and increasing length of spike trains. The performance of the first algorithm deteriorate much less than the second algorithm in case the number of neurons in the network is increased. On the other hand, the second algorithm has a better performance when the length of the spike trains are longer. In a sample dataset of 70 trains with length of average 75 spikes, both of these implementations run hundreds of times faster than the CPU in matter of performance both with regard to number of neurons and length of the spike trains.

The Cox method, previously requiring a long run-time even with an enormous computation power, is now runnable on CUDA-supported GPUs in personal computers. This implementation will certainly be useful for wide range of researchers in the field of neuronal network, neuroscience as well as neurobiology labs. Functional connectivity analysis makes it possible to observe and recreate networks structure directly inspired from natural structures, similar to Hierarchical Temporal Memory [70] or for reinforcement learning [71]. On the other hand, use of biological neuronal networks for computation tasks would provide an enormous improvement in cybernetics and in-terms of energy saving. Such solutions would be available as biological feed-forward neural network have been developed [72].

REFERENCES

[1] M. Kocaoglu, D. Malak, and O. B. Akan, “Fundamentals of green communications and computing: Modeling and simulation,” *Computer (Long. Beach. Calif).*, vol. 45, pp. 40–46, 2012.

[2] A. Tero, S. Takagi, T. Saigusa, K. Ito, D. P. Bebber, M. D. Fricker, K. Yumiki, R. Kobayashi, and T. Nakagaki, “Rules for biologically inspired adaptive network design.,” *Science*, vol. 327, pp. 439–442, 2010.

[3] K. Warwick, “Implications and consequences of robots with biological brains,” *Ethics Inf. Technol.*, vol. 12, pp. 223–234, 2010.

[4] J. L. Krichmar and F. Röhrbein, “Value and reward based learning in neurorobots,” *Frontiers in Neurorobotics*, vol. 7. 2013.

[5] D. Malak, M. Kocaoglu, and O. B. Akan, “Communication theoretic analysis of the synaptic channel for cortical neurons,” *Nano Commun. Netw.*, vol. 4, pp. 131–141, 2013.

[6] S. Balasubramaniam, S. Ben-Yehuda, S. Pautot, A. Jesorka, P. Lio’, and Y. Koucheryavy, “A review of experimental opportunities for molecular communication,” *Nano Commun. Netw.*, vol. 4, pp. 43–52, 2013.

[7] W. van Eck and M. H. Lamers, “Hybrid biological-digital systems in artistic and entertainment computing,” *Leonardo*, vol. 46, no. 2, pp. 151–158, 2013.

[8] T. DeMarse, A. Cadotte, P. Douglas, P. He, and V. Trinh, “Computation within cultured neural networks,” in *Engineering in Medicine and Biology Society, 2004. IEMBS’04. 26th Annual International Conference of the IEEE*, 2004, vol. 2, pp. 5340–5343.

[9] T. B. DeMarse, D. A. Wagenaar, A. W. Blau, and S. M. Potter, “The neurally controlled animat: Biological brains acting with simulated bodies,” *Auton. Robots*, vol. 11, pp. 305–310, 2001.

[10] M. Taketani and M. Baudry, *Advances in network electrophysiology: using multi-electrode arrays*. 2006.

[11] S. M. Potter, D. A. Wagenaar, R. M. R. Madhavan, and T. B. DeMarse, “Long-term bidirectional neuron interfaces for robotic control, and in vitro learning studies,” *Proc. 25th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. (IEEE Cat. No.03CH37439)*, vol. 4, 2003.

[12] S. Marom, A. Gal, C. Zrenner, V. Lyakhov, G. Shahaf, and D. Eytan, “Order-Based Representation in Networks of Cortical Neurons,” in *6th International Meeting on SubstrateIntegrated Micro Electrode Arrays*, 2008.

[13] A. M. Shultz, S. Lee, T. B. Shea, and H. A. Yanco, “Control of a Robot Arm with Artificial and Biological Neural Networks,” in *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[14] T. B. DeMarse and K. P. Dockendorf, “Adaptive flight control with living neuronal networks on microelectrode arrays,” in *Proceedings of the International Joint Conference on Neural Networks*, 2005, vol. 3, pp. 1548–1551.

[15] D. K. Welsh, D. E. Logothetis, M. Meister, and S. M. Reppert, “Individual neurons dissociated from rat suprachiasmatic nucleus express independently phased circadian firing rhythms.,” *Neuron*, vol. 14, pp. 697–706, 1995.

[16] J. L. Novak and B. C. Wheeler, “Multisite hippocampal slice recording and stimulation using a 32 element microelectrode array.,” *J. Neurosci. Methods*, vol. 23, pp. 149–159, 1988.

[17] U. Egert, B. Schlosshauer, S. Fennrich, W. Nisch, M. Fejtl, T. Knott, T. Müller, and H. Hämmerle, “A novel organotypic long-term culture of the rat hippocampus on substrate-integrated multielectrode arrays,” *Brain Res. Protoc.*, vol. 2, pp. 229–242, 1998.

[18] A. Tscherter, M. O. Heuschkel, P. Renaud, and J. Streit, “Spatiotemporal characterization of rhythmic activity in rat spinal cord slice cultures.,” *Eur. J. Neurosci.*, vol. 14, pp. 179–190, 2001.

[19] P. Darbon, C. Yvon, J. C. Legrand, and J. Streit, “INaP underlies intrinsic spiking and rhythm generation in networks of cultured rat spinal neurons,” *Eur. J. Neurosci.*, vol. 20, pp. 976–988, 2004.

[20] Y. Feld, M. Melamed-Frank, I. Kehat, D. Tal, S. Marom, and L. Gepstein, “Electrophysiological modulation of cardiomyocytic tissue by transfected fibroblasts expressing potassium channels: A novel strategy to manipulate excitability,” *Circulation*, vol. 105, pp. 522–529, 2002.

[21] K. Shimono, F. Brucher, R. Granger, G. Lynch, and M. Taketani, “Origins and distribution of cholinergically induced beta rhythms in hippocampal slices.,” *J. Neurosci.*, vol. 20, pp. 8462–8473, 2000.

[22] K. Egashira, K. Nishii, K. I. Nakamura, M. Kumai, S. Morimoto, and Y. Shibata, “Conduction abnormality in gap junction protein connexin45-deficient embryonic stem cell-derived cardiac myocytes,” in *Anatomical Record - Part A Discoveries in Molecular, Cellular, and Evolutionary Biology*, 2004, vol. 280, pp. 973–979.

[23] M. S. Masud and R. Borisyuk, “Statistical technique for analysing functional connectivity of multiple spike trains,” *J. Neurosci. Methods*, vol. 196, pp. 201–219, 2011.

[24] T. Berry, F. Hamilton, N. Peixoto, and T. Sauer, “Detecting connectivity changes in neuronal networks,” *J. Neurosci. Methods*, vol. 209, pp. 388–397, 2012.

[25] K. Kavukcuoglu, P. Sermanet, Y.-L. Boureau, K. Gregor, M. Mathieu, and Y. LeCun, "Learning Convolutional Feature Hierarchies for Visual Recognition," *Adv. neural Inf. Process. Syst. 23*, no. 1, pp. 1090–1098, 2010.

[26] J. Martens, "Generating Text with Recurrent Neural Networks," *Neural Networks*, vol. 131, no. 1, pp. 1017–1024, 2011.

[27] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing Machines," *arXiv Prepr. arXiv1410.5401*, 2014.

[28] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Bull. Math. Biol.*, vol. 52, pp. 25–71, 1990.

[29] R. B. Stein, "Some models of neuronal variability.," *Biophys. J.*, vol. 7, pp. 37–68, 1967.

[30] B. Ermentrout, "Type I membranes, phase resetting curves, and synchrony.," *Neural Comput.*, vol. 8, pp. 979–1001, 1996.

[31] E. M. Izhikevich and E. M. Izhikevich, "Simple model of spiking neurons.," *IEEE Trans. Neural Netw.*, vol. 14, pp. 1569–72, 2003.

[32] E. M. Izhikevich, *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*, vol. First. 2007.

[33] E. M. Izhikevich and G. M. Edelman, "Large-scale model of mammalian thalamocortical systems.," *Proc. Natl. Acad. Sci. U. S. A.*, vol. 105, pp. 3593–3598, 2008.

[34] J. Lisman and N. Spruston, "Questions about STDP as a general model of synaptic plasticity," *Front. Synaptic Neurosci.*, pp. 1–5, 2010.

[35] D. O. Hebb, "The organization of behavior: a neuropsychological theory," *Sci. Educ.*, vol. 44, p. 335, 1949.

[36] K. D. Miller, "Synaptic economics: competition and cooperation in synaptic plasticity," *Neuron*, vol. 17, no. 3, pp. 371–374, 1996.

[37] E. Oja, "A simplified neuron model as a principal component analyzer.," *J. Math. Biol.*, vol. 15, no. 3, pp. 267–273, 1982.

[38] "E. Oja, Oja learning rule, Scholarpedia. 3 (2008) 3612."

[39] T. D. Sanger, "Optimal unsupervised learning in a single-layer linear feedforward neural network," *Neural Networks*, vol. 2, no. 6. pp. 459–473, 1989.

[40] A. Björck, "Solving linear least squares problems by Gram-Schmidt orthogonalization," *Bit Numer. Math.*, vol. 7, no. 1, pp. 1–21, 1967.

[41] P. J. Sjöström, *Spike-timing dependent plasticity*. Frontiers Media SA, 2012.

[42] G. Q. Bi and M. M. Poo, “Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type.,” *J. Neurosci.*, vol. 18, no. 24, pp. 10464–10472, 1998.

[43] L. Li, I. M. Park, S. Seth, J. C. Sanchez, and J. C. Príncipe, “Functional connectivity dynamics among cortical neurons: A dependence analysis,” *IEEE Trans. Neural Syst. Rehabil. Eng.*, vol. 20, no. 1, pp. 18–30, 2012.

[44] A.-H. Shapira and I. Nelken, “Binless Estimation of Mutual Information in Metric Spaces,” *Spike Timing Mech. Funct.*, p. 121, 2013.

[45] L. Li, I. Park, S. Seth, J. C. Sanchez, and J. C. Príncipe, “Neuronal functional connectivity dynamics in cortex: An MSC-based analysis,” in *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC’10*, 2010, pp. 4136–4139.

[46] I. H. Stevenson, J. M. Rebesco, L. E. Miller, and K. P. Körding, “Inferring functional connections between neurons,” *Current Opinion in Neurobiology*, vol. 18, no. 6. pp. 582–588, 2008.

[47] D. H. Perkel, G. L. Gerstein, and G. P. Moore, “Neuronal spike trains and stochastic point processes. II. Simultaneous spike trains.,” *Biophys. J.*, vol. 7, pp. 419–440, 1967.

[48] D. Brillinger, “Nerve cell spike train data analysis: a progression of technique,” *J. Am. Stat. …*, vol. 87, pp. 260–271, 1992.

[49] M. S. Bartlett, *An introduction to stochastic processes, with special reference to methods and applications*. CUP Archive, 1978.

[50] D. R. Brillinger, H. L. Bryant, and J. P. Segundo, “Identification of synaptic interactions,” *Biol. Cybern.*, vol. 22, no. 4, pp. 213–228, 1976.

[51] A. M. Aertsen, G. L. Gerstein, M. K. Habib, and G. Palm, “Dynamics of neuronal firing correlation: modulation of ‘effective connectivity’.,” *J. Neurophysiol.*, vol. 61, pp. 900–917, 1989.

[52] “A. Seth, ‘Granger causality,’ Scholarpedia, vol. 2, no. 7, p. 1667, 2007.”

[53] S. Eldawlatly, R. Jin, and K. G. Oweiss, “Identifying functional connectivity in large-scale neural ensemble recordings: a multiscale data mining approach.,” *Neural Comput.*, vol. 21, no. 2, pp. 450–477, 2009.

[54] B. Staude, S. Rotter, and S. Grün, “CuBIC: Cumulant based inference of higher-order correlations in massively parallel spike trains,” *J. Comput. Neurosci.*, vol. 29, pp. 327–350, 2010.

[55] F. Hamilton, T. Berry, N. Peixoto, and T. Sauer, “Real-time tracking of neuronal network structure using data assimilation,” *Phys. Rev. E*, pp. 1–6, 2013.

[56] T. L. T. M. K. K. Francois Christophe Vafa Andalibi, *Survey and evaluation of neural computation models for bio-integrated systems*. Elsevier, 2015.

[57] A. M. Shultz, “Modeling of the networking and activity of cultured mouse neurons for simulated experiments,” Master’s thesis, University of Massachusetts Lowell, 2013.

[58] A. Fournier, D. Fussell, and L. Carpenter, “Computer rendering of stochastic models,” *Communications of the ACM*, vol. 25. pp. 371–384, 1982.

[59] G. S. P. Miller, “The definition and rendering of terrain maps,” *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4. pp. 39–48, 1986.

[60] J. Erickson, A. Tooker, Y. C. Tai, and J. Pine, “Caged neuron MEA: A system for long-term investigation of cultured neural network connectivity,” *J. Neurosci. Methods*, vol. 175, no. 1, pp. 1–16, 2008.

[61] “Microelectrode Array (MEA) Manual [online] Available:http://www.multichannelsystems.com/sites/multichannelsystems.com/files/documents/manuals/MEA_Manual.pdf [Accessed 30th Aug 2015].”

[62] Z. C. Chao, D. J. Bakkum, and S. M. Potter, “Region-specific network plasticity in simulated and living cortical networks: comparison of the center of activity trajectory (CAT) with other statistics,” *J. Neural Eng.*, vol. 4, no. 3, p. 294, 2007.

[63] A. Bruzzone, V. Pasquale, P. Nowak, J. Tessadori, P. Massobrio, and M. Chiappalone, “Interfacing in Silico and in Vitro Neuronal Networks,” *IEEE Eng. Med. Biol. Soc.*, 2015.

[64] K. Lenk and B. Priwitzer, “INEX--A binary neuronal model with inhibitory and excitatory synapses,” *BMC Neurosci.*, vol. 12, no. Suppl 1, p. P260, 2011.

[65] D. S. Kahng, Y. Nam, and D. Lee, “Stochastic simulation model for patterned neural multi-electrode arrays,” in *Proceedings of the 7th IEEE International Conference on Bioinformatics and Bioengineering, BIBE*, 2007, pp. 736–740.

[66] R. Segev and E. Ben-Jacob, “Generic modeling of chemotactic based self-wiring of neural networks,” *Neural Networks*, vol. 13, no. 2, pp. 185–199, 2000.

[67] F. M. Gafarov, “Self-wiring in neural nets of point-like cortical neurons fails to reproduce cytoarchitectural differences.,” *J. Integr. Neurosci.*, vol. 5, no. 2, pp. 159–169, 2006.

[68] R. Segev, M. Benveniste, Y. Shapira, and E. Ben-Jacob, “Formation of electrically active clusterized neural networks.,” *Phys. Rev. Lett.*, vol. 90, no. 16, p. 168101, 2003.

[69] T. P. Patel, S. C. Ventre, and D. F. Meaney, “Dynamic changes in neural circuit topology following mild mechanical injury in vitro,” *Ann. Biomed. Eng.*, vol. 40, no. 1, pp. 23–36, 2012.

[70] J. Hawkins and D. George, “Hierarchical temporal memory: Concepts, theory and terminology,” 2006.

[71] T. H. Teng, A. H. Tan, and J. M. Zurada, “Self-Organizing Neural Networks Integrating Domain Knowledge and Reinforcement Learning,” *IEEE Transactions on Neural Networks and Learning Systems*, 2014.

[72] A. Natarajan, T. B. DeMarse, P. Molnar, and J. J. Hickman, “Engineered In Vitro Feed-Forward Networks,” *J. Biotechnol. Biomater.*, vol. 03, no. 01, pp. 1–7, 2013.

APPENDIX A1.SOURCE CODE OF THE COX METHOD

This appendix provides the source code for the entire method. This information can be used to reproduce experiments and results presented in the paper.

```
def cox (nn,maxi, target,tsp,delta):
    whole = datetime.now()
    p = nn-1
    if p == 1 :
        gamma0 = 0.95
    else:
        gamma0 = 1 - (1-0.05)/ (p*(p-1))
    if gamma0 < 0.95:
        gamma0 = 0.95
    pval = 1 - gamma0
    tol = 0.0001*ones((p))
    flag = 1
    tspa = target
    isi = target[1:] - target [:len(target)-1]
    v = zeros([p,len(tspa)])
    v1 = zeros ([p,len(tspa)])
    la = []

    for i in range (0,p):
        index = where((tspa-delta[i])>0)[0]
        k = min(index)
        start = tspa[k] - delta [i]
        isia = append(start,isi[k:])
        la = append(la,len(isia))
        tspam = cumsum(isia)
        v[i,0:la[i]] = isia [0:la[i]]
        v1[i,0:la[i]]= tspam [0:la[i]]

    laf = min (la)
    isiat = v [0:p,0:laf]
    tspamt = v1 [0:p, 0:laf]
    b = zeros(p)
    tspz = append(b,tsp )
    tspz = reshape(tspz, (maxi+1,p))
    inda = zeros_like(isiat)
    a = zeros_like(isiat)
    for i in range (0,p):
        inda [i,:] = sort (isiat[i,:])
        atmp  =  [[ii  for  (v,  ii)  in  sorted((v,  ii)  for  (ii,  v)  in
enumerate(isiat[i]))]]
        a[i,:] = array(atmp)
      mod = SourceModule("""
<<<ALG.1/ALG.2 KERNEL>>>
      """)
      mod2 = SourceModule("""
            <<<HESSIAN KERNEL>>>
      """)
      func = mod.get_function("z_function")
    tspamt =tspamt.astype(float32)
    inda = inda.astype(float32)
    a = a.astype(float32)
    isiat = isiat.astype(float32)
    tspz = tspz.astype(float32)
    b = zeros((p,laf,laf))
    z = b.astype(float32)
    tspamt_d = tspamt
    inda_d = inda
    a_d = a
    isiat_d = isiat
    tspz_d = tspz
    p_d = p-1
    maxi_d = maxi
    laf_s = int_(sqrt(laf)+1)
    start = datetime.now()
func(cuda.InOut(tspamt_d),cuda.InOut(inda_d),cuda.InOut(a_d),cuda.In-
Out(isiat_d),cuda.InOut(tspz_d),cuda.In-
Out(z),int32(p_d),int64(maxi_d),block   =   (laf_s,laf_s,1),   grid   =
(p,int_(laf)))
    end = datetime.now()
    ztime= end-start
    print(ztime)
    bet = 0.2*ones(p)
    landa = 1 ;
    for i in range (0,100):
        scc = zeros_like(z) ;
        for l in range (0,p):
            scc [l,:,:] = bet[l] * z[l,:,:]
        ssum = zeros((laf,laf))
        for g in range (0,p):
            ssum = ssum + scc[g,:,:]
        sumte = sum(tril(exp(ssum)),axis=0)
        score = zeros((p))
        for n in range (0,p):
            temp             =             sum(divide(sum(tril(multi-
ply(z[n,:,:],exp(ssum))),axis = 0),sumte))
            score[n] = trace(z[n,:,:])-temp
        vi = zeros ((p,p));
        vi =vi.astype(float32)
        laf_d = laf.astype(int32)
        z2 = z.astype(float32)
        func2 = mod2.get_function("hess")
        ssum_d = exp(ssum)
        ssum_d= ssum_d.astype(float32)
        sumte_d = sumte.astype(float32)

func2(cuda.InOut(float32(z2)),cuda.InOut(ssum_d),cuda.InOut(sumte_d),
int32(laf_d),int32(p),cuda.InOut(vi),block = (p,1,1) ,grid = (p,1,1))
        dot_temp = dot(vi.T,vi)
        estimate  =  bet  +  reshape(dot(linalg.inv(vi),reshape(score,
(p,1))),(1,p))[0]
        if i == 0:
            initial_score = zeros_like(score)
        if i > 1:
            if linalg.norm(score)<linalg.norm(initial_score):
                landa = landa/2
            else:
                landa = landa*2
        initial_score = score
        dif_temp = abs(bet-estimate)
        if ((dif_temp< tol).all()):
            bet_result = estimate
            flag = 0
            break
        bet = estimate
    if (flag==1):
        bet_result = 100000
        betahat = 1000000
        betaci = [1000000,1000000]
    else:
        betahat = bet_result
    x = norm.ppf(1-pval/2)
    nx = [-x,x]
    betaci = zeros((p,2))

    for i in range (0,p):
        betaci[i,0] = betahat[i] + nx[0] / sqrt(vi[i,i])
        betaci[i,1] = betahat[i] + nx[1] / sqrt(vi[i,i])
    whole_end = datetime.now() - whole
    print (" the whole is" , whole_end)
    return (betahat, betaci,ztime)
```

Program 3. *The source code for the entire method*

APPENDIX A2.CUDA KERNEL FOR ALG.1

This appendix shows kernel part (CUDA – part handled by GPU) of the code for the first algorithm.

```
__global__ void z_function(float *tspamt, float *inda, float *a, float
*isiat, float *tspz,  float *z, int *p_d , int *maxi_d)
{
    float gm = 0.0955 ;
    float alphas = 10 ;
    float alphar = 0.1 ;
    float t1;
    int m = threadIdx.y + threadIdx.x * blockDim.y;
    int i = blockIdx.y;
    int j = blockIdx.x;
    int maxi = (int) maxi_d + 1;
    int p = (int)p_d +1;

    if (i>=j)
        {
          int temp = a[m*gridDim.y+i];
          int temp2 = a[m*gridDim.y+j];
          int index = 0 ;
          t1 = tspamt [m*gridDim.y+temp] - isiat[m*gridDim.y+temp] +
isiat[m*gridDim.y+temp2] ;

for (int k = m; k < p*maxi ;k+=p)
    {
     if (tspz [k] < t1 && tspz [k] != -1)
       {
          if (index < k)
            {
              index= k ;
             }
        }
    }
float bwt;
bwt = t1 - tspz [index];
z[gridDim.y*gridDim.y*m  +  gridDim.y*i  +  j]  =  (1/gm)*((exp(-bwt/al-
phas)-exp(-bwt/alphar)) /(alphas-alphar));
    }
}
```

Program 4. *Kernel part (CUDA – part handled by GPU) of the code for the first algorithm*

APPENDIX A3.CUDA KERNEL FOR ALG. 2

This appendix shows kernel part (CUDA – part handled by GPU) of the code for the second algorithm.

```
__global__ void z_function(float *tspamt, float *inda, float *a, float
*isiat, float *tspz,  float *z, int *p_d , int *maxi_d)
{
float gm = 0.0955 ;
float alphas = 10 ;
float alphar = 0.1 ;
float t1;
int m = blockIdx.x;
int i = blockIdx.y;
int j = threadIdx.y + threadIdx.x * blockDim.y;;
int maxi = (int) maxi_d + 1;
int p = (int)p_d +1;

if (i>=j)
            {
int temp = a[m*gridDim.y+i];
int temp2 = a[m*gridDim.y+j];
int index = 0 ;
t1  =  tspamt  [m*gridDim.y+temp]  -  isiat[m*gridDim.y+temp]  +  isiat
[m*gridDim.y+temp2] ;
for (int k = m; k < p*maxi ;k+=p)
        {
if (tspz [k] < t1 && tspz [k] != -1)
{
if (index < k)
{
index= k ;
}
}
}
float bwt;
bwt = t1 - tspz [index];
z[gridDim.y*gridDim.y*(blockIdx.x)  +  gridDim.y*i+j]  =  (1/gm)*((exp(-
bwt/alphas)-
exp(-bwt/alphar))/(alphas-alphar));
}
   }
```

Program 5. *Kernel part (CUDA – part handled by GPU) of the code for the second algorithm.*

APPENDIX A4.HESSIAN KERNEL

This appendix presents a kernel for accelerating the computation of the Hessian of ML, i.e. the second order differential of ML.

```
__global__  void  hess(float  *z2,float  *ssum_d,float  *sumte_d  ,int
laf,int p, float *vi)
{
    int m = threadIdx.x ;
    int n = blockIdx.x ;
    float temp1 = 0;
    float temp2 = 0;
    float temp3 = 0;
    float part1 = 0;
    float part2 = 0;
    float part3 = 0;
    float part4 = 0;

    for (int j = 0; j<laf ;j++)
    {
for (int i = j; i<laf*laf; i += laf )
            {
temp1 +=   z2[m*laf*laf + i] * z2[n*laf*laf + i] * ssum_d[i];
                  temp2 += z2[m*laf*laf + i] * ssum_d[i];
                  temp3 += z2[n*laf*laf + i] * ssum_d[i];
}
    part1 += temp1/sumte_d[j];
    part2 += temp2 ;
    part3 += temp3 ;
    part4 += (temp2*temp3)/ (sumte_d[j]*sumte_d[j]);
    temp1 = 0;
    temp2 = 0;
    temp3 = 0;
    }
vi[threadIdx.x * gridDim.x + blockIdx.x] =
part1-part4;
}
```

Program 6. *Kernel for accelerating the computation of the Hessian of ML*

APPENDIX B. INVERSED TWO-LEVEL DIAMOND SQUARE ALG.

This appendix, presents the code for inversed two-level diamond square algorithm.

```
__author__ = 'V_AD'
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt
import numpy as np
from numpy import *
import random
import visual
import math as mt
import matplotlib.pyplot as plt
from PIL.BmpImagePlugin import o32
from killer import killer

global r, stop_flag
stop_flag = False
def locations (gridsize,neuronsize,totalcell):
#     gridsize = raw_input('Enter the MEA area in micrometers^2: ')
#     neuronsize = raw_input ('Enter the neuron area in microme-
ters^2: ')
#     neuronnumber = raw_input ('Enter the total number of neurons :
')
    needed_cells = gridsize / neuronsize
    w = int(mt.sqrt(needed_cells))
    h = w
    global r,stop_flag
    r = -3
    def diamond(w,h,rand = -3,draw = False):
        global r,stop_flag
        def plasma(x, y, width, height, c1, c2, c3, c4):
            newWidth = width / 2
            newHeight = height / 2
            global idx,dots
            if (width > gridSize or height > gridSize):
                #Randomly displace the midpoint!
                midPoint = (c1 + c2 + c3 + c4) / 4 + Displace(rand)
                #Calculate the edges by averaging the two corners of
each edge.
                edge1 = (c1 + c2) / 2
                edge2 = (c2 + c3) / 2
                edge3 = (c3 + c4) / 2
                edge4 = (c4 + c1) / 2

                #Do the operation over again for each of the four new
grids.
                plasma(x, y, newWidth, newHeight, c1, edge1, mid-
Point, edge4)
                plasma(x + newWidth, y, newWidth, newHeight, edge1,
c2, edge2, midPoint)
                plasma(x + newWidth, y + newHeight, newWidth,
newHeight, midPoint, edge2, c3, edge3)
                plasma(x, y + newHeight, newWidth, newHeight, edge4,
midPoint, edge3, c4)
            else:
                #This is the "base case," where each grid piece is
less than the size of a pixel.
                c = (c1 + c2 + c3 + c4) / 4
    #             dots[idx] = c
    #             idx = idx + 1
        #         print(c)
                if (c>0.5):
                    c = 0
                    if (draw == True):
                        visual.points(pos=[x-(100),c,y-(100)],
color=(1,0.31,0.1))
                    dots[idx] = c
                    idx = idx + 1
                else:
                    c= 5
                    if (draw == True):
                        visual.points(pos=[x-(100),c,y-(100)],
color=(0.8,1,1))
                    dots[idx] = c
                    idx = idx + 1

        def Displace(num):
            rand = (random.uniform(num, 1) - noise)
        #     print rand
            return rand


        global gridSize, gamma, points, width, height, idx,dots
        random.seed('Albert Einstein was a German theoretical physi-
cist.')

        def reshaper (inpt):
            l= len(inpt)
            if (l>4):
                o1= inpt[0:l/4]
                o2= inpt[l/4:l/2]
                o3= inpt[l/2:3*l/4]
                o4= inpt[3*l/4:]
                temp1 = np.concatenate((reshaper(o1),re-
shaper(o2)),axis=1)
                temp2 = np.concatenate((reshaper(o4),re-
shaper(o3)),axis=1)
                temp3 = np.concatenate((temp1,temp2),axis = 0)
                return (temp3)
            else:
                outp = np.zeros((2,2))
                outp[0,0:2] = inpt [0:2]
                outp[1,0:2] = np.fliplr([inpt[2:]])[0]
                return(outp)

        nearest_2_power = 0
        while w>2**nearest_2_power:
            nearest_2_power += 1
        nearest_2 = 2**nearest_2_power
        idx = 0
        width = nearest_2*10
        noise = 0.02 # less noise = higher map
        height = nearest_2*10
        gridSize = 10 # size between pixels
        length = 0
        gridtemp = width
        while gridtemp > gridSize:
            gridtemp = gridtemp/2
            length +=1
#         print(length)
        dots = np.zeros (4**length)

        plasma(0,0, width, height, random.uniform(0, 1), random.uni-
form(0, 1), random.uniform(0, 1), random.uniform(0, 1))
    #     print(dots)
        positions = np.zeros
((mt.sqrt(len(dots)),mt.sqrt(len(dots))))

        positions = reshaper(dots)
    #     print (positions)
        cells = sum(sum(positions==5))
#         print (cells,'out of', len(positions)**2)
        return (positions,cells)
    position_final , cells = diamond(w, h)
    position_final_cropped = position_final [0:w,0:w]
    cells = sum(sum(position_final_cropped==5))

    while cells > totalcell:

        r += 0.1
        position_final , cells = diamond (w,h,r)
        position_final_cropped = position_final [0:w,0:w]
        cells = sum(sum(position_final_cropped==5))
        # print(cells)
    else:
        r -= 0.1
        position_final , cells = diamond (w,h,r)
        position_final_cropped = position_final [0:w,0:w]
        cells = sum(sum(position_final_cropped==5))

#     else:
# #         if (stop_flag == False):
# #             stop_flag = True
# #         position_final , cells = diamond (w,h,r,True)
#         position_final , cells = diamond (w,h,r,True)
#         position_final_cropped = position_final [0:w,0:w]
#         cells = sum(sum(position_final_cropped==5))
    # else:
    #     r = rand-0.1
    #     locations (w,h,r)

    print (cells,'out of', len(position_final_cropped)**2)
    q = position_final_cropped
    eliminate  = np.random.randint(cells,size = cells-totalcell)
    all_indices = where(q==5)
    for el in eliminate :
        q[all_indices[0][el]][all_indices[1][el]] = 0
    for i in range (0,len(q)):
        for j in range (0,len(q)):
            if (q[i,j]==0):
                visual.points(pos=[i,0,j], color=(0.8,1,1))
            else:
                visual.points(pos=[i,1,j], color=(1,0.31,0.1))
    p_temp = np.where(position_final_cropped == 5)
    p = [array([i[1],j[1]]) for i in list (enumerate(p_temp[0])) for
j in list(enumerate(p_temp[1])) if i[0] == j[0]]
    p = [array([pp[1]*(sqrt(neuronsize)),pp[0]*(-sqrt(neuronsize)) ])
for pp in p]
    return p,position_final_cropped



def MEA_locations (gridsize,neuronsize,totalcell):
    p,q = locations(gridsize,neuronsize,totalcell)
    ax = plt.subplot(111)
    # ax.plot ([-i[1] for i in p],[-i[0] for i in p],'ro')
    real_size = 50
    # origp = np.copy(p)
    # length = sqrt(neuronsize)
    for loc in p :
        i0,i1 = np.random.random(2) * (sqrt(neuronsize))
        loc[0] += i0
        loc[1] -= i1

    return p,q
# p,q = locations(50000,5000, 4)
p,q = MEA_locations(7840000,5000,1000)

# p2,_ = MEA_locations(7840000,50,1000)
# print (q)
# print(p)
ax = plt.subplot(111)
ax.plot ([-i[1] for i in p],[-i[0] for i in p],'ro')

ax.set_ylim([-2800,0])
ax.set_xlim([0,2800])

plt.show()
# print(np.shape(q))
# p = killer(q,0)
# for i in range (0,len(p)):
#         for j in range (0,len(p)):
#             if (p[i,j]==0):
#                 visual.points(pos=[i,0,j], color=(255/255, 255/255,
255/255))
#             elif (p[i,j]==2.5):
#                 visual.points(pos=[i,1,j], color=(51/255, 102/255,
200/255))
#             else:
#                 visual.points(pos=[i,1,j], color=(255/255 ,204/255
,10/255))
# print (q)
# for i in range (0,len(q)):
#     for j in range (0,len(q)):
#         if (q[i,j]==0):
#             visual.points(pos=[i,0,j], color=(0.8,1,1))
#         else:
#             visual.points(pos=[i,1,j], color=(1,0.31,0.1))
```

Program 7. *The code for inversed two-level diamond square algorithm used for plating*

APPENDIX C. CELL DEATH

This appendix, presents the code for implementing cell death.

```
__author__ = 'V_AD'
import numpy as np
import random as rand
import matplotlib.pyplot as plt

def death_list (neurons):
    NN = len(neurons)
    output = np.zeros(12240);
    q = float(NN)
    percentage = float(rand.sample(range(45,55),1)[0])/100
    x = rand.sample(range(0,12240),int(q*percentage))
    y = rand.sample(neurons,int(q*percentage))
    output[x] = y
    return output

def killer (q,days):
    NN = np.count_nonzero(q)
    neurons = np.zeros (NN)
    l = len (q)
    counter = 0
    for i in range (0,l):
        for j in range (0,l):
            if q[i,j]!=0:
                idx = i*l+j
                neurons[counter] = idx
                counter+=1
    dead_idx = death_list(neurons)
    dead_idx = dead_idx [0:days*60*12]

    for i in range (0,len(dead_idx)):
        if dead_idx[i] !=0:
            dead = dead_idx[i]
            x = dead/l
            y = dead%l
            q[x,y] = 2.5
    return q
```

Program 8. *Implementation of cell death*

APPENDIX D. MORPHOLOGY

This appendix provides the code for generating the distributions based on morphology.

```
from shapely.geometry import LineString
from matplotlib.pyplot import *
from locations import *
from numpy import *
import numpy as np
import numpy as np
from rotator import *
import matplotlib.pyplot as plt
import math as mth
import time
def morpho_generator (area, neuron , n, morpho_source ) :
    p,q = MEA_locations(area,neuron, n)
    print (p)
    # print morpho_source [5]
    total_morpho = zeros ([n,shape(morpho_source)[0],shape(mor-
pho_source)[1]])
    morpho_temp = copy(morpho_source)
    for r in range (n):
        soma_x = float(morpho_temp[0][2]) + p[r][0]
        soma_y = float(morpho_temp[0][3]) + p[r][1]
        angle = np.random.random(1) * 360
        for l in range(len(morpho_temp)):
            temp_x = float(morpho_temp[l][2]) + p[r][0]
            temp_y = float(morpho_temp[l][3]) + p[r][1]
            point  = rotator((soma_x,soma_y),(temp_x,temp_y),angle)
            # morpho_temp[l][2] = str(float(morpho_temp[l][2]) +
p[r][0])
            # morpho_temp[l][3] = str(float(morpho_temp[l][3]) +
p[r][1])
            morpho_temp[l][2] = str(point[0])
            morpho_temp[l][3] = str(point[1])
            # l[2] = str(float(l[2]) + p[r][1])
            # l[3] = str(float(l[3]) + p[r][0])
        total_morpho [r,:,:] = morpho_temp
        # print morpho_temp[0]
        morpho_temp = copy(morpho_source)

    # print morpho_source[5]
    return total_morpho , p,q



def multi_morpho_generator (area, neuron , n, morpho_source ,types,
dist ) :

    p,q = MEA_locations(area,neuron, n)
    print (p)
    # print morpho_source [5]
    pick = array ([])
    for idx , qq in list(enumerate(dist)):
        temp_ = round(qq*n)
        pick = append(pick, (idx)*ones(temp_)) # this creates an ar-
ray of 1 , 2, ... each repeated by number of cells per type. in next
```

```
step it will be shuffled to determine the final formation of all
type of cells.
    np.random.shuffle(pick)
    while len(pick) < n :
        pick = append (pick, array([len(dist)]))
        random.shuffle(pick)
    total_morpho = {}
    # total_morpho = zeros ([n,shape(morpho_source)[0],shape(mor-
pho_source)[1]])
    # morpho_temp = copy(morpho_source)


    for r in range (n):
        morpho_temp = copy(morpho_source[types[int(pick[r])]]) #
this put the right model inside the temp
        # morpho_temp = [map(float,p) for p in morpho_temp]
        soma_x = float(morpho_temp[0][2]) + p[r][0]
        soma_y = float(morpho_temp[0][3]) + p[r][1]
        angle = np.random.random(1) * 360
        for l in range(len(morpho_temp)):
            temp_x = float(morpho_temp[l][2]) + p[r][0]
            temp_y = float(morpho_temp[l][3]) + p[r][1]
            point  = rotator((soma_x,soma_y),(temp_x,temp_y),angle)
            # morpho_temp[l][2] = str(float(morpho_temp[l][2]) +
p[r][0])
            # morpho_temp[l][3] = str(float(morpho_temp[l][3]) +
p[r][1])
            morpho_temp[l][2] = str(point[0])
            morpho_temp[l][3] = str(point[1])

            # l[2] = str(float(l[2]) + p[r][1])
            # l[3] = str(float(l[3]) + p[r][0])
        total_morpho [r] = {}
        total_morpho [r]['type'] = types[int(pick[r])]
        total_morpho [r]['points'] = [map(float,pp) for pp in mor-
pho_temp]

        # total_morpho [r,:,:] = morpho_temp
        # print morpho_temp[0]
        # morpho_temp = copy(morpho_source)

    # print morpho_source[5]
    return total_morpho , p,q , pick

def rotator(centerPoint,point,angle):
    """Rotates a point around another centerPoint. Angle is in de-
grees.
    Rotation is counter-clockwise"""
    angle = math.radians(angle)
    temp_x = point[0]-centerPoint[0]
    temp_y = point[1] - centerPoint [1]
    dist = sqrt(temp_x**2 + temp_y**2)
    # old_ang = arccos(temp_x/dist)
    # total_ang = old_ang + angle
    new_x = float('%.2f'%(cos(angle)*temp_x - sin(angle)*temp_y +
centerPoint[0]))
    new_y = float('%.2f'%(sin(angle)*temp_x + cos(angle)*temp_y +
centerPoint [1]))
```

```
    # temp_point = point[0]-centerPoint[0] , point[1]-centerPoint[1]
    # temp_point = ( temp_point[0]*math.cos(angle)-
temp_point[1]*math.sin(angle) , temp_point[0]*math.sin(an-
gle)+temp_point[1]*math.cos(angle))
    # temp_point = temp_point[0]+centerPoint[0] , temp_point[1]+cen-
terPoint[1]
    temp_point = array([new_x,new_y])
    return temp_point

def all_permutations (temp_axons,temp_dends):
    results = array([])
    for a in range(1,len(temp_axons)) :
        if float(temp_axons[a][0]) == float(temp_axons[a][6])+1:
            temp_line1 = LineString([(float(temp_ax-
ons[a][2]),float(temp_axons[a][3])),(float(temp_axons[a-
1][2]),float(temp_axons[a-1][3]))])
        else:
            target_idx = [p[0] for p in zip(*where(temp_axons == ar-
ray([temp_axons[a][6]]))) if p[1] == 0][0]
            temp_line1 = LineString([(float(temp_ax-
ons[a][2]),float(temp_axons[a][3])),(float(temp_axons[tar-
get_idx][2]),float(temp_axons[target_idx][3]))])
        for d in range (1,len(temp_dends)):
            if float(temp_dends[d][0]) == float(temp_dends[d][6])+1:
                temp_line2 = Lin-
eString([(float(temp_dends[d][2]),float(temp_dends[d][3])),(float(te
mp_dends[d-1][2]),float(temp_dends[d-1][3]))])
            else:
                target_idx = [p[0] for p in zip(*where(temp_dends ==
array([temp_dends[d][0]]))) if p[1] == 0][0]
                temp_line2 = Lin-
eString([(float(temp_dends[d][2]),float(temp_dends[d][3])),(float(te
mp_dends[target_idx][2]),float(temp_dends[target_idx][3]))])
        temp_line = temp_line1.intersection(temp_line2)
        if temp_line :
            results = (results,array(temp_line1.intersec-
tion(temp_line2)))
    return len(results) , results # this function returns the number
of synapses between a set of dends which are totlaly inside a set a
axons as well as the points of connection


######################################################################
##############



def partial_permutation (temp_axons , temp_dends ):
    results = array([])
    for a in range(1,len(temp_axons)) :
        temp_line1 = LineString([(float(temp_ax-
ons[a][2]),float(temp_axons[a][3])),(float(temp_axons[a-
1][2]),float(temp_axons[a-1][3]))])
        for d in range (1,len(temp_dends)):
            temp_line2 = Lin-
eString([(float(temp_dends[d][2]),float(temp_dends[d][3])),(float(te
mp_dends[d-1][2]),float(temp_dends[d-1][3]))])
            temp_line = temp_line1.intersection(temp_line2)
            if temp_line :
                results = (results,array(temp_line1.intersec-
tion(temp_line2)))
    return len(results[1:]) , results[1:]

def status (str):
    cleaner = ' ' * 100
    print '\r'+ cleaner + '\r' + str,
####################################################

NN =100
neuron_size = 5000

plate_size = 851929
# neuron = array(['1', '1', '2.39', '-1.6', '-563.94', '11.1795', '-
1'])
types = ['pyramidal','basket']
types_location = ['C:/Users/admin_tunnus/Desktop/pyrami-
dal.swc','C:/Users/admin_tunnus/Desktop/basket.swc' ]
neurons_source = {}
for i in range (len(types)) :
    neurons_source[types[i]] =  array(['1', '1', '2.39', '-1.6', '-
563.94', '11.1795', '-1'])
    with open (types_location[i], 'r') as f :
        for line in f:
            if  not (line.startswith('#')):
                neurons_source[types[i]] = vstack ((neu-
rons_source[types[i]], line.split()))
    neurons_source[types[i]] = neurons_source[types[i]][1:]
# neuron = neuron [1:]
dist = array([0.7,0.3])
all_neurons,somas, structure , type_array = multi_morpho_genera-
tor(plate_size,neuron_size,NN,neurons_source,types,dist)
# somas = divide (somas,neuron_size)
axons = {}
dends = {}
points = {}
for idx in range (NN):
    axons["a%d" %idx] = [p for p in all_neurons[idx]['points'] if
p[1]==2]
    dends["d%d" %idx] = [p for p in all_neurons[idx]['points'] if
p[1]==3]
    points["n%d" %idx] = {}
# total_max = max(max([ qq[2] for pp in all_neurons for qq in pp ]),
max([ q[3] for p in all_neurons for q in p ]))
# total_min = min(min([ qq[2] for pp in all_neurons for qq in pp ]),
min([ q[3] for p in all_neurons for q in p ]))
print "starting finding the branches"
all_branches = branch_finder (all_neurons)
print"branches are ready "

colors = array(['b','g','r','c','m','y','k'])

fig = plt.figure()
ax = fig.add_subplot(111)
print "plotting the neurons"
branch_status = 0
branch_total = len([ne for ne in all_branches])
indices_array = type_array
```

```
for pp in range(len (types)):
    temp_ar = zeros ([len(type_array)])
    counter = 0
    for ii, q in list(enumerate(type_array)) :
        if q == pp :
            temp_ar[ii] = counter
            counter +=1
    indices_array = vstack([indices_array,temp_ar])
indices_array =indices_array[1:]
final_result = {}
# points = array([0,0])
# passed_number = 0.
# sh1 = shape ([axons[p] for p in axons])
# sh2 = shape ([dends[p] for p in dends])
# total_number = sh1[0] * (sh1[1]-1) * (sh2[0]-1) * (sh2[1]-1)
# print (total_number)
fr = array([])
to = array([])
sy = array([])

print "finding boundry boxes"
axon_borders = zeros ([NN,4,2]) # border points of each neuron 0 is
upper left, 1 is upper right and so on
dend_borders = zeros ([NN,4,2])
counter1 = 0
for neuron in range (NN):
    axon_borders[neuron,0,:],axon_borders[neuron,1,:],axon_bor-
ders[neuron,2,:], axon_borders[neuron,3,:] = boundry_box (ax-
ons['a%d'%neuron])
    dend_borders[neuron,0,:] , dend_borders[neuron,1,:] , dend_bor-
ders[neuron,2,:], dend_borders[neuron,3,:] =  boundry_box
(dends['d%d'%neuron])
    counter1+= 1
    percentage = float(counter1)/NN *100
    status ("%.2f %% of locating borders completed\n" %percentage)
print (structure)
# print (axon_borders)
# print(dend_borders)
### next two lines draw the whole boundries
plt.plot([qq[0] for pp in axon_borders for qq in pp],[qq[1] for pp
in axon_borders for qq in pp],'ro')
plt.plot([qq[0] for pp in dend_borders for qq in pp],[qq[1] for pp
in dend_borders for qq in pp],'bs')
# show()
# raw_input("press to continue...")
dig_level = 1
zeros_src = 0
syn_set = -1

connection_map = {}


print "start finding axons and dends borders"
for q,t in zip(dend_borders,axon_borders) :
    col = random.choice(colors)
    for t2 in range (3) :
        plt.plot ( [t[t2][0],t[t2+1][0]],[t[t2][1],t[t2+1][1]],col)
        plt.plot([q[t2][0],q[t2+1][0]],[q[t2][1],q[t2+1][1]],col )
    plt.plot ( [t[0][0],t[3][0]],[t[0][1],t[3][1]],col)
    plt.plot ( [q[0][0],q[3][0]],[q[0][1],q[3][1]],col)
# show()

print "creating connection map "
map_current = 0
map_total = NN
for targ in range (NN) :
    temp_axon_border = axon_borders[targ,:,:]
    connection_map['n%d'%targ] = {}
    connection_map['n%d'%targ]['neuron_type'] = type_array[targ]
    for ref in range (NN):
        if targ!=ref :
            temp_dends_border = dend_borders[ref,:,:]
            _,temp_type,temp_boundry = intersect_finder
(temp_axon_border[0],temp_axon_border[1],temp_axon_bor-
der[2],temp_axon_border[3],temp_dends_border[0],temp_dends_bor-
der[1],temp_dends_border[2],temp_dends_border[3],boundry_flag=1)
            if temp_type != '0P':
                connection_map['n%d'%targ]['n%d'%ref]= {}
                connection_map['n%d'%targ]['n%d'%ref]['type'] =
temp_type
                connection_map['n%d'%targ]['n%d'%ref]['neuron_type']
= type_array[ref]
                connection_map['n%d'%targ]['n%d'%ref]['boundry'] =
temp_boundry
    map_current +=1
    map_perc = float (map_current *100 ) / map_total
    status ("%.2f%% of creating connection map completed"%map_perc)


# show( )
# print (connection_map)
# raw_input("somehting")
# show()
# print connection_map


total_calc =  len([q for p in connection_map for q in connec-
tion_map[p] if q!=  'neuron_type'])
current_calc = 0
print "Finding Connections"
for src , target in list(enumerate(connection_map)):
    for dest, reference in list(enumerate(connection_map[target])):
        if reference != 'neuron_type' :
            print target,reference
            start_time = time.time()
            temp_boundries_boundry = connection_map[target][refer-
ence]['boundry']
            temp_type = connection_map[target][reference]['type']
            print temp_type
            temp_axons_branches = [all_branches[target]['axons'][b]
for b in all_branches[target]['axons'] if intersect_finder\
                (all_branches[target]['ax-
ons'][b]['boundry'][0],all_branches[target]['ax-
ons'][b]['boundry'][1],\
                 all_branches[target]['ax-
ons'][b]['boundry'][2],all_branches[target]['ax-
ons'][b]['boundry'][3],\
temp_boundries_boundry[0],temp_boundries_boundry[1],temp_boundries_b
oundry[2],temp_boundries_boundry[3])[1] != '0P' ]
            temp_dends_branches = [all_branches[refer-
ence]['dends'][b] for b in  all_branches[reference]['dends'] if in-
tersect_finder\
                (all_branches[refer-
ence]['dends'][b]['boundry'][0],all_branches[refer-
ence]['dends'][b]['boundry'][1],\
                 all_branches[refer-
ence]['dends'][b]['boundry'][2],all_branches[refer-
ence]['dends'][b]['boundry'][3],\

temp_boundries_boundry[0],temp_boundries_boundry[1],temp_boundries_b
oundry[2],temp_boundries_boundry[3])[1] != '0P' ]
            syn_set+=1
            final_result['syn_set%d'%syn_set] = {}
            final_result['syn_set%d'%syn_set]['from'] = {}
            final_result['syn_set%d'%syn_set]['from']['idx'] = tar-
get
            final_result['syn_set%d'%syn_set]['from']['type'] =
connection_map[target]['neuron_type']
            final_result['syn_set%d'%syn_set]['to'] = {}
            final_result['syn_set%d'%syn_set]['to']['idx'] = refer-
ence
            final_result['syn_set%d'%syn_set]['to']['type'] =connec-
tion_map[target][reference]['neuron_type']
            final_result['syn_set%d'%syn_set]['n'] = 0
            final_result['syn_set%d'%syn_set]['points'] = array
([0,0]) # remeber to remove first element since it's an empty array
            total_per = len(temp_axons_branches) *
len(temp_dends_branches)
            current_per = 0
            for t_ax in temp_axons_branches :
                for t_de in temp_dends_branches:
                    if (inter-
sect_finder(t_ax['boundry'][0],t_ax['boundry'][1],t_ax['boundry'][2]
,t_ax['boundry'][3],t_de['boundry'][0],t_de['boundry'][1],t_de['boun
dry'][2],t_de['boundry'][3])[1] != '0P'):
                        temp_final_n  ,temp_final_points = par-
tial_permutation([q1 for q1 in t_ax['points']],[q2 for q2 in
t_de['points']])
                        final_result['syn_set%d'%syn_set]['n'] +=
temp_final_n
                        if temp_final_n != 0 :
                            final_re-
sult['syn_set%d'%syn_set]['points'] =  vstack ([final_re-
sult['syn_set%d'%syn_set]['points'],temp_final_points])
                        # status ("%d found"%final_re-
sult['syn_set%d'%syn_set]['n'])
                    current_per +=  1
                    current_perc= float(current_per)*100/total_per
                    status("%.2f%% of current neuron is finnished
"%current_perc)
            elapsed_time = time.time() - start_time
            print (" the time is : %f " %elapsed_time)
            final_result['syn_set%d'%syn_set]['points'] = final_re-
sult['syn_set%d'%syn_set]['points'][1:]
            current_calc += 1
            total_calc_perc=  float(current_calc)*100 / total_calc
            status ("#################### Totally %.2f%% completed" 
%total_calc_perc)
print(final_result)

# following three lines gather the points of connection and plot 
them with green triangles
for_plot1 = [i for i in [final_result[p]['points'] for p in fi-
nal_result] if len(i)!= 1]
for_plot = [qq for jj in for_plot1 for qq in jj]
plt.plot([i[0] for i in for_plot],[i[1] for i in for_plot],'b^')
plt.plot([i[0] for i in for_plot],[i[1] for i in for_plot],'y^')
#########
# save('C:/Users/andalibi/Local/connectionmap', connection_map)
# save('C:/Users/andalibi/Local/final_result', final_result)


pickle.dump( connection_map, open( "C:/Users/admin_tunnus/Desk-
top/results/connectionmap.p", "wb" ) )
pickle.dump( final_result, open( "C:/Users/admin_tunnus/Desktop/re-
sults/final_result.p", "wb" ) )
pickle.dump( indices_array, open( "C:/Users/admin_tunnus/Desktop/re-
sults/indices_array.p", "wb" ) )
pickle.dump( all_branches, open( "C:/Users/admin_tunnus/Desktop/re-
sults/all_branches.p", "wb" ) )
pickle.dump( all_neurons, open( "C:/Users/admin_tunnus/Desktop/re-
sults/all_neurons.p", "wb" ) )
pickle.dump( somas, open( "C:/Users/admin_tunnus/Desktop/results/so-
mas.p", "wb" ) )
pickle.dump( structure, open( "C:/Users/admin_tunnus/Desktop/re-
sults/structure.p", "wb" ) )
plt.axis('equal')
show()

def branch_finder (all_neurons ) :
    branches = {}
    finder_current = 0
    finder_total = len(all_neurons)
    for idx in range (len(all_neurons)):
        branches ["n%d"%idx] = {}
        branches ["n%d"%idx]['axons'] = {}
        branches ["n%d"%idx]['dends'] = {}
        # temp_1 = [p for p in all_neurons[idx] if ((p[0]!=p[6]+1) 
and len([q for q in all_neurons[idx] if q[0]==p[6]])!=0)]
        temp_1 = [p for p in all_neurons[idx]['points'] if 
((int(p[0])!=int(p[6])+1) and len([q for q in all_neu-
rons[idx]['points'] if q[0]==p[6]])!=0)]
        all_0 = array([int(j[0]) for j in all_neu-
rons[idx]['points']])
        places = [where (all_0 == int(jj[0])) for jj in temp_1]
        counter1 = 0 # this is for axons
        counter2 = 0 # this is for dendrites
        for m1 in range (1,len(temp_1)):
            start_idx = int(places[m1][0]) # start is the first 
point of branch , begin is the 0 point of branch
            prev_start_idx = int(places[m1-1][0])
            begin_idx = int(all_neurons
[idx]['points'][prev_start_idx][6]) -1
```

```
            if int(all_neurons[idx]['points'][start_idx][1]) == 2:#
this goes for axonal brnaches
                branches ['n%d'%idx]['axons']['br%d'%counter1] = {}
                branches ['n%d'%idx]['axons']['br%d'%coun-
ter1]['points'] = vstack([all_neurons
[idx]['points'][begin_idx],all_neurons[idx]['points'][prev_start_idx
: start_idx]]) # the branch is created and the first point is also
considered
                branches ['n%d'%idx]['axons']['br%d'%coun-
ter1]['boundry'] = boundry_box (branches ['n%d'%idx]['ax-
ons']['br%d'%counter1]['points'])
                counter1+=1
            elif int(all_neu-
rons[idx]['points'][int(places[m1][0])][1]) == 3:
                branches ['n%d'%idx]['dends']['br%d'%counter2] = {}
                branches ['n%d'%idx]['dends']['br%d'%coun-
ter2]['points'] = vstack([all_neurons
[idx]['points'][begin_idx],all_neu-
rons[idx]['points'][prev_start_idx:start_idx]])
                branches ['n%d'%idx]['dends']['br%d'%coun-
ter2]['boundry'] = boundry_box (branches
['n%d'%idx]['dends']['br%d'%counter2]['points'])
                counter2+=1
        finder_current +=1
        finder_perc = float(finder_current*100)/finder_total
        status ("%.2f%% of branch finding completed"%finder_perc)
    return branches
```

Program 9. *The implementation used for utilizing morphology for the purpose of finding connections*