Washington University in St. Louis

Washington University Open Scholarship

McKelvey School of Engineering Theses & Dissertations

McKelvey School of Engineering

Spring 5-2018

Development of Scalable Simulator for Spiking Neural Network

Jae Sang Ha
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Computer and Systems Architecture Commons

Recommended Citation

Ha, Jae Sang, "Development of Scalable Simulator for Spiking Neural Network" (2018). *McKelvey School of Engineering Theses & Dissertations*. 345.
https://openscholarship.wustl.edu/eng_etds/345

This Thesis is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS
School of Engineering and Applied Science
Department of Computer Science and Engineering

Thesis Examination Committee:
Shantanu Chakrabartty, Advisor
Xuan Zhang
Brian Kocoloski

Development of Scalable Simulator for Spiking Neural Network

by

Jae Sang Ha

A thesis presented to the School of Engineering
of Washington University in St. Louis in partial fulfillment of the
requirements for the degree of
Master of Science

May 2018

Saint Louis, Missouri

Contents

List of Figures ... iii

List of Tables ... iv

Acknowledgments (Required) ... v

Dedication (Optional) ... vi

Abstract (Optional) ... vii

1 Introduction ... 1
- 1.1 Background ... 1
- 1.2 Outline of Thesis ... 2

2 Neural Simulators ... 3
- 2.1 State-of-art of Neural Simulators ... 4
- 2.2 Research Objective ... 5

3 Spiking Neural Network ... 6
- 3.1 Spiking Neuron Mechanism ... 7
- 3.2 Biological Neuron Models ... 9
- 3.3 Growth Transform Neuron ... 10

4 Implementation ... 12
- 4.1 Software Architecture ... 12
- 4.2 Optimization ... 13
  - 4.2.1 BLAS ... 13
  - 4.2.2 sparseBLAS ... 14
- 4.3 Performance ... 15

5 Network Topology ... 17
- 5.1 Interconnection Matrix ... 17
- 5.2 Graphical User Interface ... 20
- 5.3 Coupling Matrix and Spike Raster Plot ... 21

6 Conclusion ... 24

Appendix A Code for The Simulator ... 25

References ... 54

List of Figures

Figure 1.1 Trend in Neuromorphic Engineering 1
Figure 3.1 Biological Neuron 7
Figure 3.2 Spiking Neuron 7
Figure 3.3 General Structure of Spiking Neural Network 8
Figure 4.1 Software Architecture of the Simulator 12
Figure 4.2 Execution Times for Three Different Implementations 15
Figure 5.1 Building Connection between Neurons from Two Clusters 17
Figure 5.2 Example of Network Topology for Three Clusters 18
Figure 5.3 Example of Network Topology for Six Clusters 19
Figure 5.4 Coupling Matrix Page of GUI 20
Figure 5.5 Spike Plot Page of GUI 21
Figure 5.6 Coupling Matrix and Corresponding Spike Raster Plot 22

List of Tables

Table 2.1: Number of Cerebral Cortex Neurons 3
Table 2.2: Comparison of Different Neural Simulators 4
Table 3.1: Different Neuron Models 9
Table 4.1: Specification of MacBook Pro (2016) 15

Acknowledgments

I would first like to thank my thesis advisor Dr. Shantanu Chakrabartty of the Electrical Engineering at Washington University in St. Louis. The door to Prof. Chakrabartty's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it.

I would also like to thank the experts who were involved in the validation survey for this research project: Dr. Xuan Zhang and Dr. Brian Kocoloski. Without their passionate participation and input, the validation survey could not have been successfully conducted.

I would also like to acknowledge my colleagues from my lab at Washington University in St. Louis as the listener of this thesis, and I am gratefully indebted to them for their very valuable comments on this thesis.

Finally, I must express my very profound gratitude to my family and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Jae Sang Ha

Washington University in St. Louis
May 2018

Dedicated to my family and God.

ABSTRACT OF THE THESIS

Development of Scalable Simulator for Spiking Neural Network

by

Jae Sang Ha

Master of Science in Computer Science

Washington University in St. Louis, 2018

Research Advisor: Professor Shantanu Chakrabartty

A neural network simulator for Spiking Neural Network (SNN) is a useful research tool to model brain functions with a computer. With this tool, different parameters can be explored easily compared to using a real brain. For several decades, researchers have developed many software packages and simulators to accelerate research in computational neuroscience. However, despite their advantages, different neural simulators possess different limitations, such as flexibility of choosing different neuron models and scalability of simulators for large numbers of neurons. This paper demonstrates an efficient and scalable spiking neural simulator that is based on growth transform neurons and runs on a single machine. The growth transform neuron model's update is based on matrix-vector multiplication, which is optimized using external libraries named BLAS and sparseBLAS. Using sparseBLAS, the scalability of the simulator was optimized with sparse representation of matrix. The optimized tool can simulate up to 1 million neurons and is flexible with neuron model changes behind the simulator. Furthermore, with a simple graphical user interface, a researcher can easily design a variety of network topology with different parameters. He/she can visualize a coupling matrix, simulate a designed network and study the spike train with spike raster plot. This simulator will be made open source so that researchers can benefit from this for large-scale simulations.

Chapter 1

Introduction

As people begin to study brains and analyze its activities, a lot of researchers put their efforts in developing efficient and scalable neural simulators that can contribute to computational neuroscience. This chapter gives a brief summary of the background of neuromorphic engineering and neural simulators.

1.1 Background

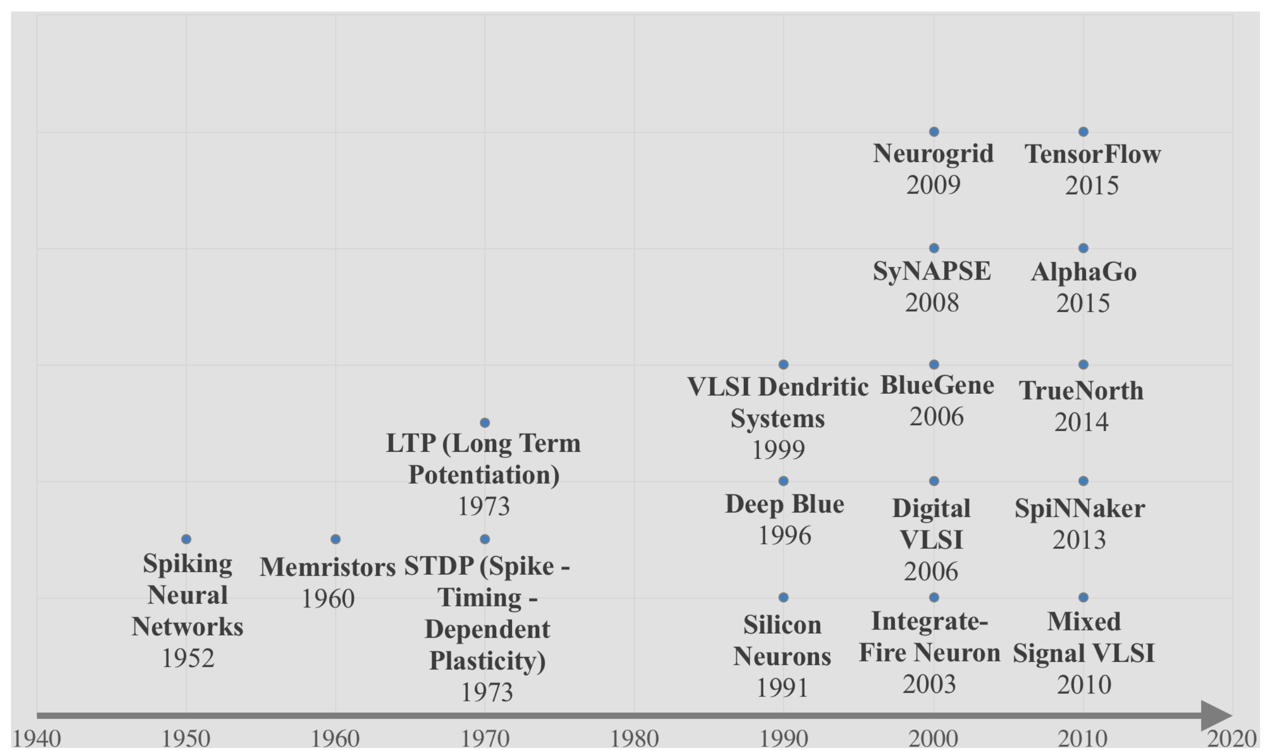


Figure 1.1 Trend in Neuromorphic Engineering

As illustrated in Figure 1.1, although the spiking neural network was developed in 1952, the research on neuromorphic engineering was not too active until the early 2000s. Recently, there has been a lot of research taking place in this field, and many large-scale neural simulators such as Neurogrid, SpiNNaker, TrueNorth were developed. These simulators take advantage of parallel computing platforms and were able to simulate more than billion number of neurons ([4]).

Before the development of the neural simulator, the only way to study the brain was through direct measurement of firing patterns and activity by optical or electrical means. Those techniques include EEG, fMRI, CT, PET, and MEG. However, many mapping techniques include hundreds of thousands of neurons in a single voxel, resulting in a relatively low resolution. Also, many functions of this equipment involve multiple parts of brain, which generally includes incorrect assumptions about how brain functions are actually divided. With this criticism, researchers found out another way to study the brain by simulating the computer model using basic information given about some components of the brain. In this kind of research, the achieved pattern from the simulation should match with what people have observed and should as accurately track the direct measurement as possible. If patterns look similar with biological recordings, a researcher can use the simulation to understand something. In the early 2000s, researchers began to focus on developing different types of neural simulators that can simulate large number of neurons efficiently ([11]).

1.2 Outline of Thesis

Chapter 2 explains in depth about usefulness of large-scale neural simulators and compare different neural simulators currently used in research. After the understanding of neural simulators, Chapter 3 explains a mechanism of spiking neurons, different neuron models and growth transform neuron, which is used in this simulator. Then, Chapter 4 reveals the software architecture of this simulator and implementation details including optimization techniques. Chapter 5 describes the network topology and important features about graphical user interface. Chapter 6 summarizes the research and end with conclusion.

Chapter 2

Neural Simulators

Neural simulators were developed to overcome the limitations of direct measurement method used to study brain activities. They are useful research tools to model brain functions by a computer, and researchers do not need to look for a real brain to experiment on. Neural simulators apply spiking behavior of brain, so they are realistic enough without sacrificing any details. Also, researchers can have freedom of exploring different kinds of parameters on a computer model and test their hypotheses of how brain works numerically leading them to easily study brain mechanisms and behaviors. Such advantages have made many research communities to build efficient neural simulators.

Table 2.1 Number of Cerebral Cortex Neurons

Fly	10^5 neurons
Mouse	4×10^6 neurons
Cat	3×10^8 neurons
Human	10^{11} neurons

Researchers soon realized they need not just a neural simulator but a scalable neural simulator. From Table 2.1, a human has about 100 billion cerebral cortex neurons in brain. Before the development of neural simulators, studying brain on a large-scale was a big concern for the recording mechanisms cannot get beyond hundreds of neurons using direct measurement. To simulate and study the scale of mouse's brain, people would need a scalable simulator that can simulate up to at least 1 million

neurons. There exist neural simulators that can take advantage of parallel computing hardware platforms and simulate up to a few billion neurons. There are Neurogrid, SpiNNaker and TrueNorth, which can simulate spiking neural networks directly in hardware using processors as a building block of a computing platform ([2]). However, since not all research communities have this ability to have those computing platforms, people also focused on developing some scalable neural simulators that can run on a single machine. For software neural simulators, there are Brian, NEURON and NEST, which will be closely analyzed in the next section.

2.1 State-of-art of Neural Simulators

Table 2.2 Comparison of Different Neural Simulators

	maximum # of neurons	advantage	limitation
Brian	4,000	- concise language - model flexibility	- small-scale simulation
NEURON	-	- large scale modelling - detailed model	- fixed set of neuron models
NEST	20 million~	- small # of compartments modelling - large network model	- fixed set of neuron models

Brian[7], NEURON[8] and NEST[9] are open source software packages for developing simulations of networks of spiking neurons. From the Table 2.2, Brian highlights its flexibility and simplicity or designing user specific neuron models; users can give mathematical equations to create their own models. Unlike Brian, NEURON and NEST have a fixed set of neuron models that users have to choose from. Brian, however, can only run on a single machine, which limits the scalability of simulations, and the researchers from Brian are currently developing new version that can run on a

GPU to improve the scalability. NEST, on the other hand, can distribute simulations across a cluster, and is capable of designing large scale networks. NEST focuses on dynamics, size and structure of neural systems, rather than the exact geometry of individual neurons, whereas NEURON is capable of designing detailed network model. Both NEURON and NEST can run on hardware platforms allowing large scalability of simulations.

2.2 Research Objective

Observing the limitations of three popular software neural simulators, I developed a scalable simulator for spiking neural networks with flexibility on model selection and simplicity of designing coupling matrices. The purpose of this research is to develop a large-scale simulator using spiking neural network with flexible neuron model selection and intuitive user interface that can design a variety of network topology. Users have freedom to change neuron models in backend by simply replacing the existing algorithm with the algorithm of their choices. This can be further studied by running different neuron models with the simulator, but, at this stage, we will focus more on scalability and the network topology of this simulator. In particular, I have chosen growth transform neuron as a model for this simulator because it is under development and has a lot of benefits. To achieve my goals, I developed simulator utilizing the libraries.

Chapter 3

Spiking Neural Network

Before the development of spiking neural networks, second generation artificial neural networks were widely used in machine learning, making lots of breakthrough progress in many fields. This generation of artificial neural networks are fully connected networks that take in continuous values and output continuous values. Unlike the first generation of neural network, which used threshold function to give digital output, this second generation of neural networks uses activation function that outputs continuous analog values. These first two generation of neural networks used rate coding, which is a coding scheme that assumes that most information about the stimulus is contained in the firing rate of neuron. This coding scheme, however, ignores any information encoded in temporal structure of the spike train and treats it as a "noise." Although the second generation, that used continuous activation function to model the intermediate output frequencies of neurons, was biologically more realistic than the previous generation, its limitation of inaccurately mimicking actual mechanism of brain's neurons motivated the development of the third generation of neural networks, which is called spiking neural networks ([10]).

3.1 Spiking Neuron

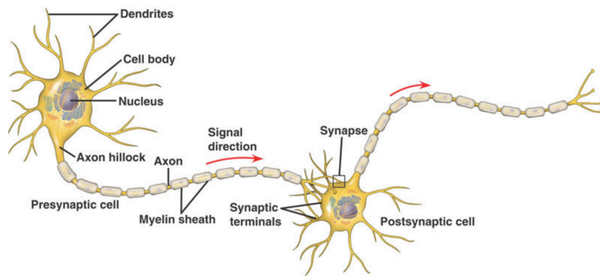


Figure 3.1 Biological Neuron

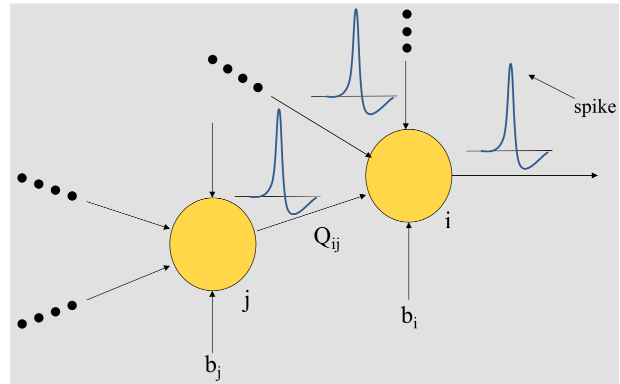


Figure 3.2 Spiking Neuron

The nervous system consists of a number of neurons. Similar to Figure 3.1([14]), each neuron has a cell body, dendrites, axon, and axon hillock. Near its end, axon is divided into several branches, each of which ends in a synaptic terminal. The site of communication between a transmitting cell (a presynaptic neuron) and a receiving one (a postsynaptic cell) is called the synapse. Dendrites receive signals that get transmitted by axons. Then, an action potential (spike) gets generated. It is generated when membrane potential of a specific axon location rapidly rises and falls, which leads to depolarization of adjacent locations. This action potential is significant in cell-to-cell communication in neurons by propagating signals. "Action potentials in neurons are also known as "nerve impulses" or "spikes", and the temporal sequence of action potentials generated by a neuron is called its "spike train". A neuron that emits an action potential, or nerve impulse, is often said to "fire"."

Spiking neural network is comprised of a number of neurons that communicate with each other generating spikes to other neurons. Before looking at the whole architecture of spiking neural network, it can be closely examined with Figure 3.2. This has a presynaptic neuron, 'j', that is transmitting spikes to the postsynaptic neuron, 'i'. Postsynaptic neuron integrates spike information from all the presynaptic neurons that it is connected with, and produces its own spikes depending on the coupling

weights, Q_{ij}. Each neuron has its own external stimulus, b_i and b_j. Like this, each neuron integrates or combines spikes from other neurons, and generates own spike. Spiking neural network consists of these spiking neurons, and its architecture is shown in Figure 3.3.

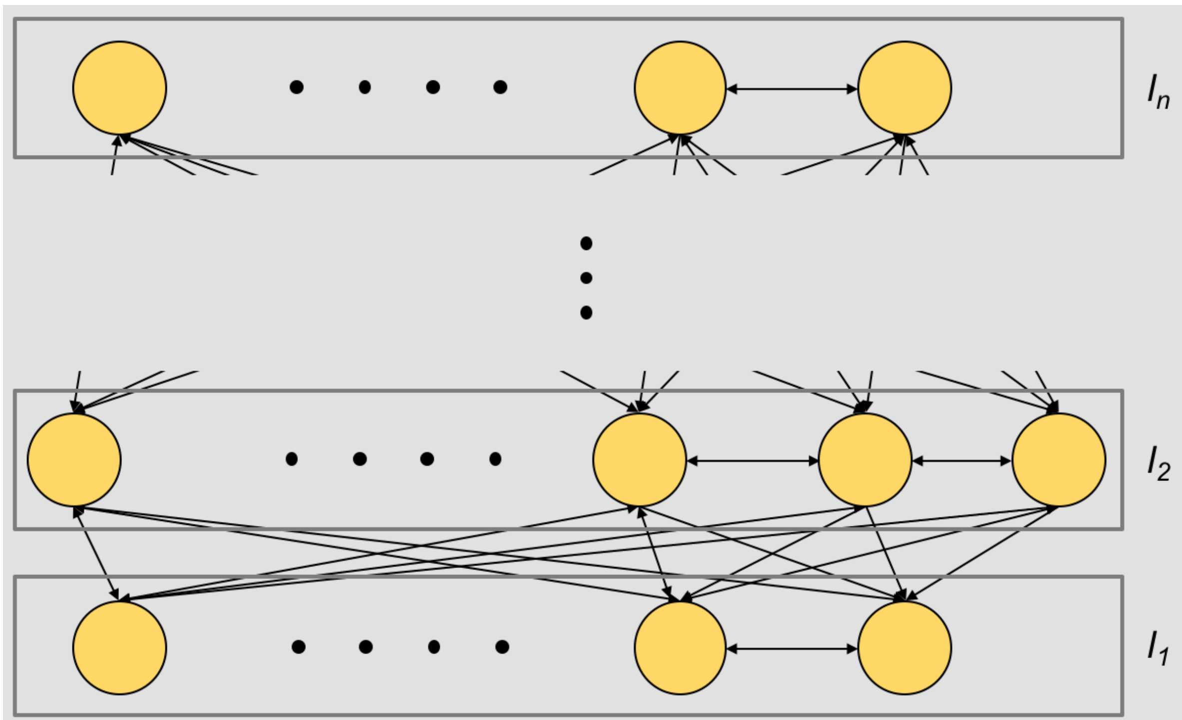


Figure 3.3 General Structure of Spiking Neural Network

Spiking neural network is important in a sense that it increased the level of realism in a neural simulation by integrating the time into the model; neurons in spiking neural network actually fire only when a membrane potential reaches a specific value, generating a signal that increases or decreases potentials of other neurons. With incoming spikes, the current activation level becomes higher, which eventually leading to either firing or decaying over time. From Figure 3.3, the general structure of spiking neural network is described as neurons in each layer are interconnected and are also connected with neurons from next layer. However, everything may not be interconnected for other structures can also have sparse connections between neurons. Due to its realistic properties, lots of applications

began to use this model. It was used in information processing, operation of biological neural circuits, and so on. However, due to its increased computational costs with simulating realistic neural models, it was more useful in neuroscience rather than engineering. Despite this limitation, it was applied in both software and hardware for simulation ([15]).

3.2 Biological Neuron Models

Based on many different experimental settings and difficulty to separate intrinsic properties of a single neuron, many neuron models have appeared throughout the research in this field. These models include Hodgkin-Huxley[12] model, Izhikevish[6] model, Leaky Integrate-and-Fire[13] model, and so on.

Table 3.1 Different Neuron Models

Hodgkin-Huxley model	- Can simulate all neuro-computational properties of biological spiking neurons - Expensive to implement (only small number of neurons)
Izhikevish model	- Can exhibit firing patterns of all known types of cortical neurons - Quite efficient in large-scale simulations of cortical networks
Leaky Integrate-and-Fire model	- Simplest model to implement using one or two variables - Not accurate model for simulation
Growth Transform Neuron model	- connects the spiking dynamics to a network-level objective function that minimize the error - gives intuition to explore neuromorphic system from machine learning point of view since typical machine

	learning algorithms learn based on minimization of an error

The leaky integrate-and-fire (LIF) neuron, being one of the earliest models of a neuron, is one of the mostly used in computational neuroscience. It is easy to implement because it has much less number of variables compared to other neuron models. Due to its simplicity, it cannot simulate most of neuro-computational properties of biological spiking neurons, being one of the worst models to use in simulations. Unlike LIF, the Izhikevish model can simulate firing patterns of all known types of cortical neurons with four parameters. It is also scalable in efficient simulations of cortical networks. The Hodgkin-Huxley model, consisting of ten parameters, not only allows researchers to investigate many questions related to a single cell dynamic but also has biophysically meaningful and measurable parameters, being one of the most important models in computational neuroscience. With many number of parameters to estimate or measure, this is not easily tractable with complex systems of neurons. Since it is extremely expensive to implement, it is usually used to simulate a small number of neurons.

3.3 Growth Transform Neuron

For the simulator, the specific neuron model, called growth transform neuron[1], is used. This model is developed in my institution's lab and possesses some properties that distinguishes itself from other neuron models. This neuron model is "tightly coupled to a system objective function, which results in network dynamics that are always stable and interpretable; and the process of spike generation and population dynamics is the result of minimizing an energy functional." It connects the spiking dynamics to a network-level objective function that tries to always minimize the error with which the inputs are represented by the neural response. It offers us an intuition to explore neuromorphic systems from a machine learning point of view, because typically machine learning algorithms learn based on the minimization of some sort of an error. The key part of this neuron model appears with the following update equation:

$$\boldsymbol{P_i^+} \leftarrow \frac{\boldsymbol{P_i^+}}{\boldsymbol{\mu_i}}(-\sum_j \boldsymbol{Q_{ij}}\boldsymbol{P_j^+} - \boldsymbol{b_i^+} - \boldsymbol{\psi}(\boldsymbol{P_i^+}) + \boldsymbol{K_i}) \tag{1.1}$$

$$\boldsymbol{P_i^-} \leftarrow \frac{\boldsymbol{P_i^-}}{\boldsymbol{\mu_i}}(-\sum_j \boldsymbol{Q_{ij}}\boldsymbol{P_j^-} - \boldsymbol{b_i^-} - \boldsymbol{\psi}(\boldsymbol{P_i^-}) + \boldsymbol{K_i}) \tag{1.2}$$

This model's update equation is in form of matrix-vector multiplication. Due to this property, the whole algorithm could be optimized using open-source external linear algebra libraries, BLAS and sparseBLAS, which will be talked later for implementation details.

Chapter 4

Implementation

For the scalable and efficient implementation of neural simulator, optimization is unavoidable task to be fulfilled. This chapter describes the software architecture of this simulator and investigates optimization techniques explaining about two external libraries. At the end, it compares the performance of original version and optimized ones by measuring execution time for each version.

4.1 Software Architecture

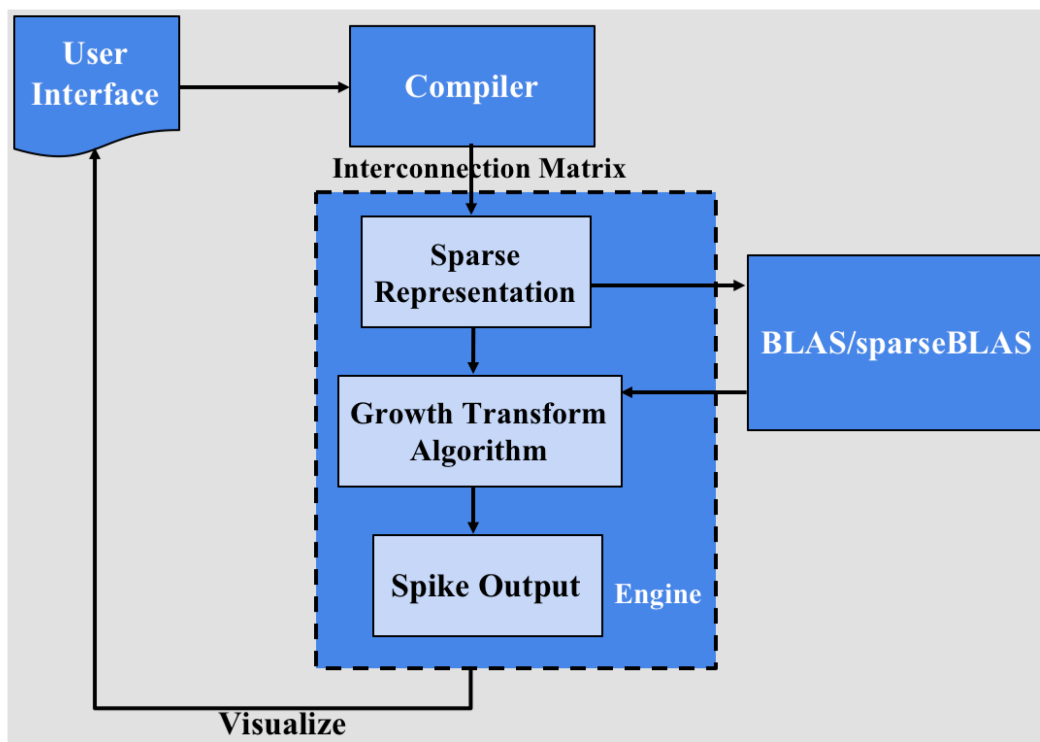


Figure 4.1 Software Architecture of the Simulator

Figure 4.1 shows the software architecture for the simulator. First, a user designs and provides the interconnection matrix with user interface, then the engine treats the generated interconnection matrix as a giant matrix. The format of the matrix is in compressed sparse row format, which the sparseBLAS takes over and runs all the optimizations for sparse matrix. After that, the algorithm in backend starts to run and generates spike patterns, which get recorded in a file. User interface finally visualizes the spike pattern as a raster plot for user to view and study the spike train.

4.2 Optimization

The growth transform neuron model update equation is based on matrix-vector multiplication, which can be optimized by using linear algebra external libraries, BLAS. Also, using sparse representation, the simulator is further optimized with another external library, sparseBLAS. This section explains the basic algorithms and optimization behind those libraries and how it was applied to the simulator.

4.2.1 BLAS

Basic Linear Algebra Subprograms (BLAS) is a "specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication." Its implementations are optimized for speed, which can bring significant performance improvements. As numerical programming become prevalent, people began to pay closer attention to the development of subroutine libraries for high-level mathematical operations. Linear algebra programs have so-called "kernel" operations, which became defined subroutines that math libraries call. The advantages of using kernel calls are more readable library routine, fewer bugs, and possible optimization for speed. BLAS had three levels of kernel operations consisting of the vector operations as level-1, matrix-vector operations as level-2, and matrix-matrix operations as level-3. BLAS make use of cache memory, which is much faster than main memory, to keep matrix manipulations localized allowing better use of the cache for matrix computations, and also used block-partitioned algorithms to improve matrix-matrix operations.

Particularly for Level 3, the linear algebra software package, LAPACK, was developed upon block-partitioned algorithms. Being a state-of-the-art software for linear algebra problems, LAPACK is widely supported by many hardware and software vendors ([17]).

4.2.2 sparseBLAS

sparseBLAS is an extension to BLAS to handle sparse matrices efficiently. As number of neurons gets larger and larger, it is easy to find sparse structure of coupling matrix, which contains most of the entries as zeros, in real world applications. Even though BLAS and LAPACK supports some sort of sparse matrix types such as band matrices and triangular matrices, irregular unstructured sparse matrices were not considered in any other libraries. After numerous research about optimization on sparse matrices, people found out that implementation of sparse matrix operations can be improved by an order of magnitude. The optimization technique behind sparseBLAS is heavily discussed in many research communities, but the general concept of optimization is using different interface from other BLAS interfaces; it uses handle-based generic interface that allows Level 2 and Level 3 operations to take a pointer to a generic representation to a created sparse matric object as an input rather than to take the matrix entries themselves. sparseBLAS provides computational kernels for sparse matrix operations such as sparse matrix products and also supports some sparse formats such as compressed-row, compressed-column, and coordinate storage formats ([16]).

4.3 Performance

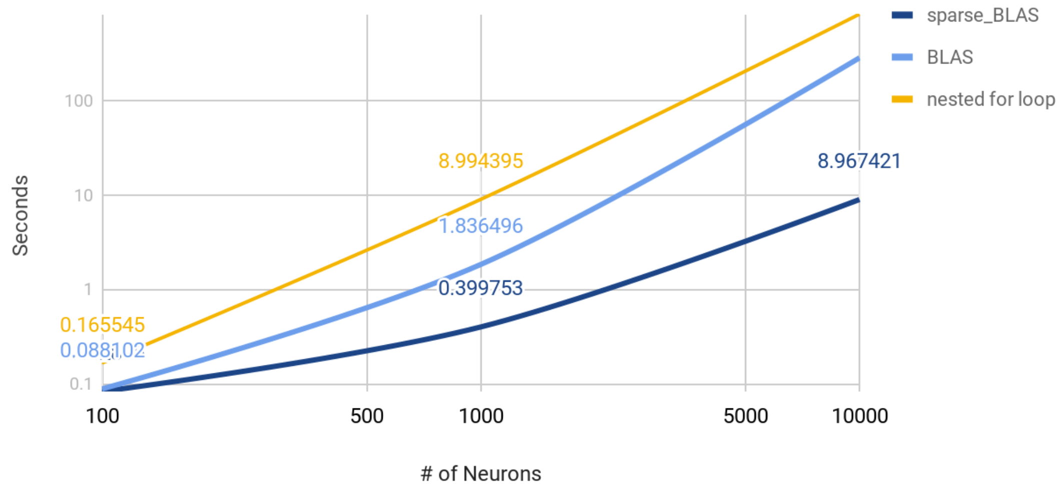


Figure 4.2 Execution Times for Three Different Implementations

Table 4.1 Specification of MacBook Pro (2016)

Operating System	macOS Sierra
Processor	2 GHz Intel Core i5
Memory	8 GB 1867 MHz LPDDR3
Flash Storage	251 GB

Having programming language in C, I compared the trivial version of code without using external libraries with versions with BLAS and sparseBLAS. With the trivial version, the matrix-vector multiplication was computed in nested for loops. The computing time could be seen in Figure 4.2.

The experiment was run on a specific interconnection matrix type. For the consistency, the sparse matrix type of 10% width, 100% density of one cluster was used as an input to the algorithm. Table 4.1 shows the specification of the PC that was used to run these simulations.

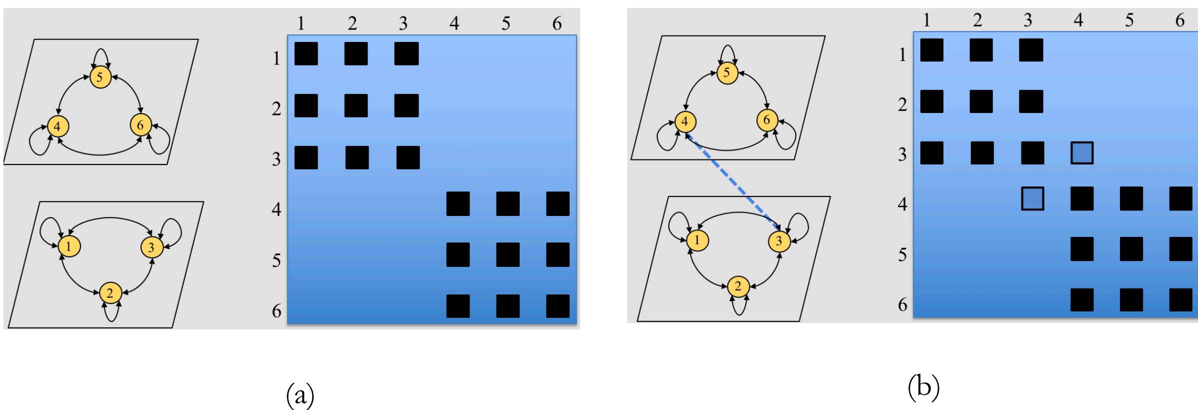
Chapter 5

Network Topology

Allowing users to design the interconnection matrix of their choice is also really important property of neural simulator. With simple and intuitive graphical user interface, a user can easily design desired network topology and study the relationship of different interconnection matrices and their spiking responses.

5.1 Design of Interconnection Matrix

For the design of interconnection matrix, users can play with four parameters including number of clusters, width (dimension) of each cluster, density of each cluster, and overlap percentage with next cluster. Each entry of the matrix, which has a size of number of neurons by number of neurons, contains a six decimal points analog weight ranging from 0 to 1.



(a) (b)

Figure 5.1 Building Connection between Neurons from Two Clusters

From the simple network topology, the intuition of different network topology can be built. Figure 5.1 visualizes how two neurons from different layer get connected. First, on (a), there are three neurons in one cluster and other three neurons in another cluster. They are numbered as 1, 2, 3, 4, 5, and 6. Neuron 1, 2, and 3 belong to the same cluster and are fully connected with each other. Neuron 4, 5, and 6 also belong to the same cluster and are fully connected. The coupling matrix for this representation consists of 9 square dots ranging from 1 to 3, and other 9 square dots ranging from 4 to 6. Now, on (b), neuron 3 and 4 are connected, and that connecting dotted line is represented with light blue square dots of the interconnection matrix indicating overlap areas of those two clusters.

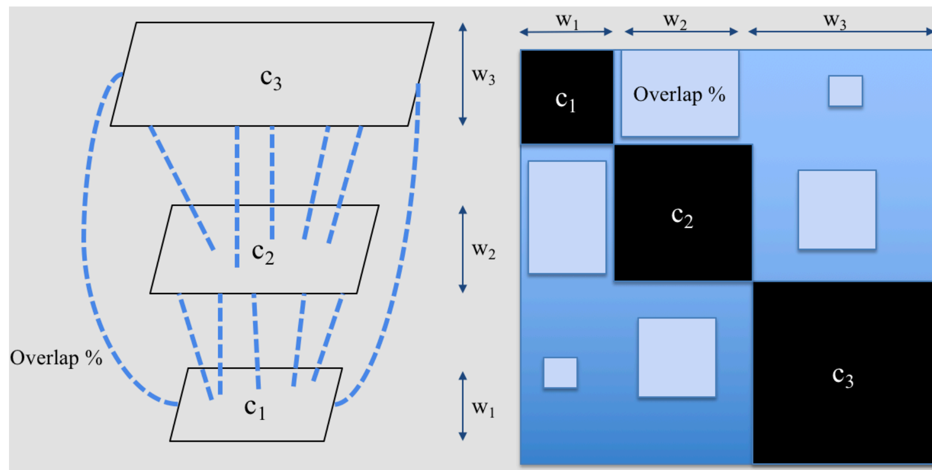


Figure 5.2 Example of Network Topology for Three Clusters

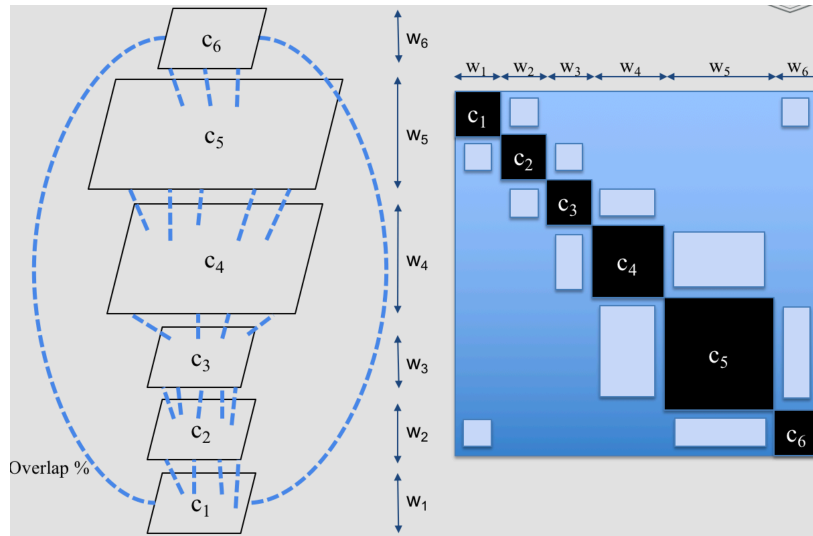


Figure 5.3 Example of Network Topology for Six Clusters

Learning from Figure 5.1, other complex network topologies such as Figure 5.2 and 5.3 can be understood. Figure 5.2 represents network topology with three clusters with different width and overlaps. User can build complex network topology such as Figure 5.3 including six layers connected with different overlap percentage, and even can build more complex networks with graphical user interface of this simulator.

5.2 Graphical User Interface

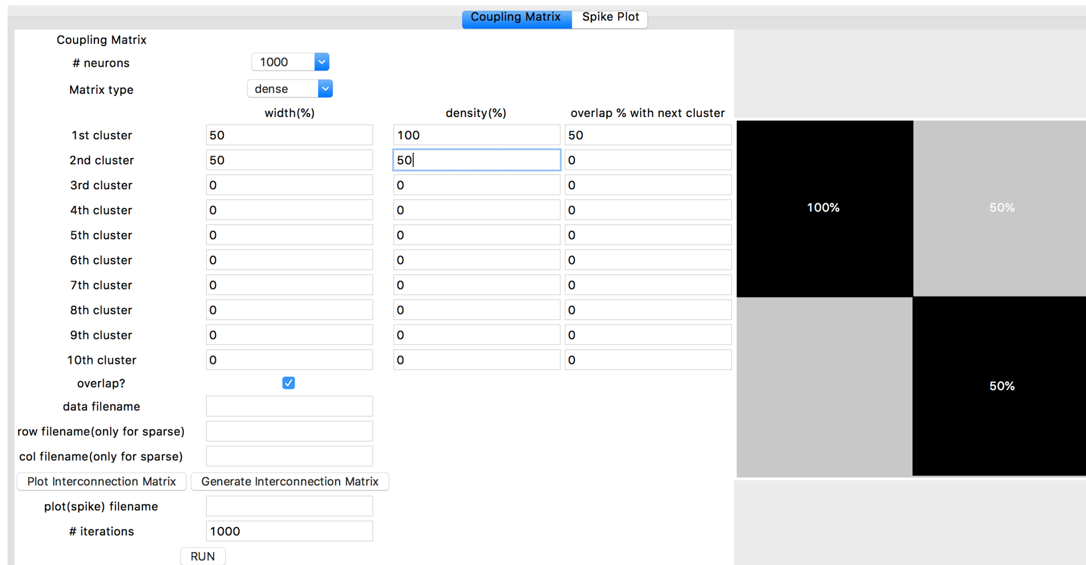


Figure 5.4 Coupling Matrix Page of GUI

This simulator's graphical user interface was built to allow users to easily design detailed interconnection matrices. Looking at figure 5.4, users have the ability to control up to 10 clusters (layers) of network including width and density of each cluster and overlap percentage with a next cluster. The number of neurons line has four options, which are 1000, 10000, 100000 and 1000000. The matrix type is either sparse or dense, and dense matrix type is only available for 1000 and 10000 number of neurons. If users want to include the overlaps with next cluster or set connectivity with next cluster, they can do it by clicking the overlap button and setting desirable values for overlap percentage. Users can view the coupling matrix before they actually generate the data file by clicking "Plot Interconnection Matrix" button besides "Generate Interconnection Matrix" button. This feature allows users to first visualize the network and, if they want to, they can create the data file for that coupling matrix. Two lines, "row filename" and "col filename", are only applicable for sparse matrix for sparse matrix uses compressed format of data file for efficiency. After users fill in data filename, row filename and col filename, they can generate the interconnection matrix and store the

matrix in file format. Then, they can run the simulation after they fill in plot filename. The number of iteration for simulation is set to 1000 as a default, but users can change this to a desirable value.

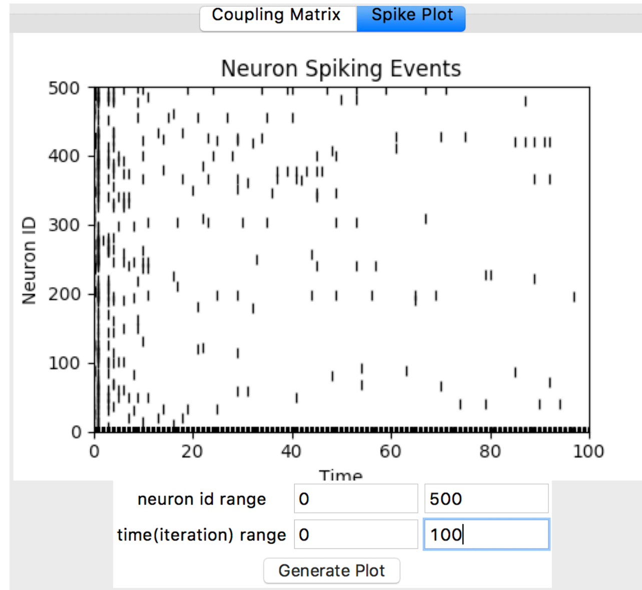


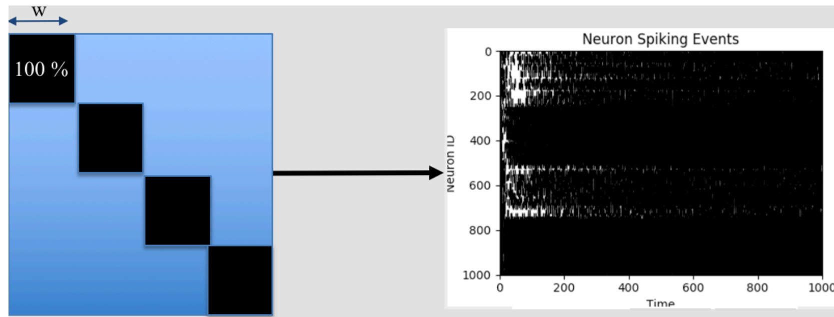
Figure 5.5 Spike Plot Page of GUI

After the simulation is done, users can move to spike plot page and study the spike patterns. The x-axis represents the number of iteration, and y-axis represents the neuron id. If a neuron id 1 had spike at 4th iteration, the marker will be put in as a coordinate of (4,1) in a raster plot. Users also have the flexibility to choose range of neuron id and time (iteration) since, as the number of neurons and number of iterations become larger and larger, it is harder to study the spike pattern with a dense raster plot. Having the freedom to study spike train of desirable range, researchers can benefit from this feature.

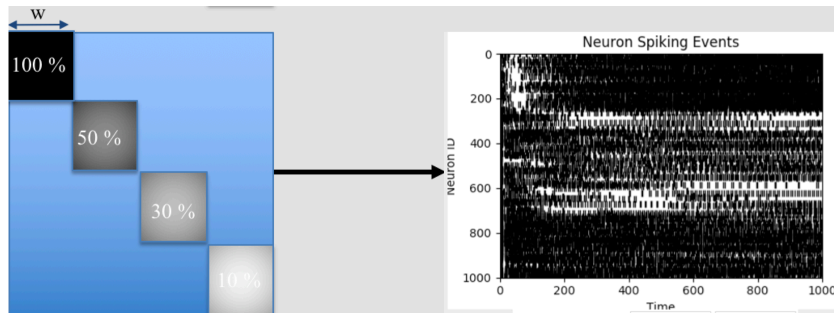
5.3 Coupling Matrix and Raster Plot

Based on the growth transform algorithm, the produced spike patterns are stored in a text file and are visualized in a raster plot in the simulator's user interface. In Figure 5.6, there are four different

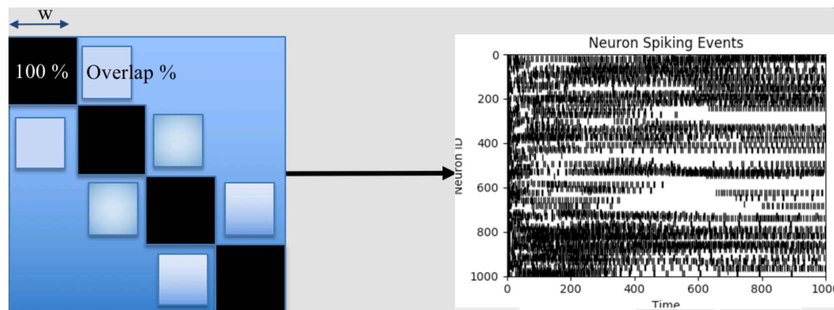
coupling matrices and corresponding spike responses. For the purpose of this comparison, I fixed the weights of four matrices for the consistency.



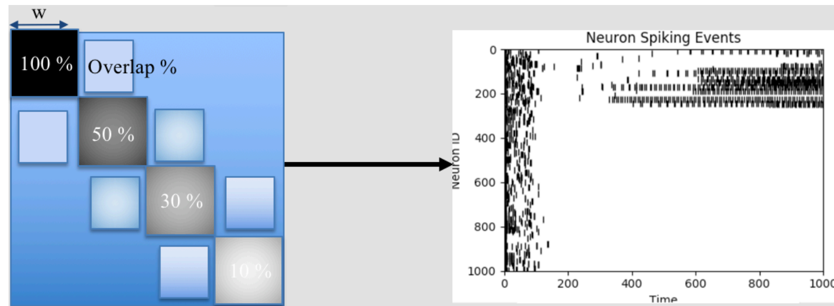
(a)



(b)



(c)



(d)

Figure 5.6 Coupling Matrix and Corresponding Spike Raster Plot

There are some interesting relationships observed. In (a) and (b), for just four clusters with no connections between other clusters, the spike pattern was dense. As the connection (overlap) between clusters were built as (c) and (d), the spike patterns look significantly different. For (c) when all four clusters fully connected matrix, the spike pattern became much sparse. For (d), where four clusters have different interconnectivity percent, the spike seems to disappear except the neurons in first cluster. The purpose of this particular research was not about studying and deriving dynamics and relationship between coupling matrix and the spiking pattern, but, as Figure 5.6 illustrates, a user has the ability to construct different network and study its behavior with spike raster plot.

Chapter 6

Conclusion

With all the literature surveys regarding neuromorphic trend and neural simulators, the development of flexible and scalable simulator with simple user interface seems reasonable. Based on spiking neural network of specific neuron model called growth transform neuron, the simulator was developed and optimized with two linear algebra libraries for efficient computation of matrix-vector operations for sparse matrix. With intuitive graphical user interface that allow users to easily build complex and detailed network topology, the simulator was advanced. Having this simulator as an open-source, the research community will have another efficient simulator to benefit from.

Appendix A

Code for The Simulator

neurons_gendata.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>

void validate_cmdline_args(int argc, int nargs, const char* msg) {
  if (argc < nargs) {
    printf("%s", msg);
    exit(EXIT_FAILURE);
  }
}

void validate_file(const char* in, const char* msg) {
  FILE* file = fopen(in, "r");
  if (file == NULL) {
    printf("%s", msg);
    exit(EXIT_FAILURE);
  }
  fclose(file);
}

int main(int argc, char* argv[]) {
  clock_t t0, t1;
  t0 = clock();
  validate_cmdline_args(argc, 5,"Usage: ./neurons_gendata <sparse?> <data-output-file-name>
<row-file> <col-file> <num_neurons> <# clusters> <1~10 cluster width> <1~10 cluster
density> <1~10 overlap %>\n");
  long long int N = atoi(argv[5]); //# of neurons
  int sparse = atoi(argv[1]);
```

```
long long int sparsity=0; //1% sparsity
if(sparse){
  sparsity=N*N*0.01; //1% sparsity
  printf("%lld\n", sparsity);
}
else{
  sparsity = N*N;
  printf("%lld\n", sparsity);
}
int clusterNum=atoi(argv[6]);
int totalWidth=0;
int clusterW[clusterNum+1];
for(int c=7; c<7+clusterNum; c++){
  if(argv[c]!=NULL){
    clusterW[c-7]=atoi(argv[c]);
    totalWidth+=clusterW[c-7];
  }
}
clusterW[clusterNum]=clusterW[0];
int clusterD[clusterNum];
for(int c=17; c<17+clusterNum; c++){
  if(argv[c]!=NULL){
    clusterD[c-17]=atoi(argv[c]);
  }
}
int overlapP[clusterNum];
for(int o=27; o<27+clusterNum; o++){
  if(argv[o]!=NULL){
    overlapP[o-27]=atoi(argv[o]);
  }
  else{
    overlapP[o-27]=0;
  }
}
long long int totalElements=0;
int clusterDim[clusterNum+1];
for(int i=0; i<=clusterNum; i++){
  if(sparse){
    clusterDim[i] = N*0.1*((clusterW[i])/100.0);
  }
```

```
  else{
   clusterDim[i] = N*((clusterW[i])/100.0);
  }
  printf("%d cluster dimension: %d\n", i, clusterDim[i]);
}
long long int density[clusterNum];
for(int i=0; i<clusterNum; i++){
  density[i]=(long long int)clusterD[i]*clusterDim[i]*clusterDim[i]*0.01;
  totalElements+=density[i];
  printf("%d cluster area: %lld\n", i, density[i]);
}
long long int overlapArea[clusterNum];
for(int i=1; i<=clusterNum; i++){
  overlapArea[i-1] = (long long int)clusterDim[i-1]*clusterDim[i]*overlapP[i-1]/100.0;
  if(overlapArea[i-1]%2==1){
   overlapArea[i-1]--;
  }
  printf("%d overlap Area: %lld\n", i-1, overlapArea[i-1]);
}
for(int i=1; i<clusterNum; i++){
  clusterDim[i] = clusterDim[i-1]+clusterDim[i];
  printf("%d cumulated dimension: %d\n", i, clusterDim[i]);
}
printf("%lld total area\n", totalElements);
int randomP = (100-totalWidth);
long long int randElements=N*0.1*randomP;
printf("%lld random area\n", randElements);
long long int num_samples = N;
double *Q = (double *) malloc(sparsity * sizeof(double));
int *x = (int *) malloc(sparsity * sizeof(int));
int *y = (int *) malloc(sparsity * sizeof(int));
long long int nz = 0;
int randrange=1000000;
int j=0;
int dimension=0;
int prevDimension=0;
int range=0;
int orange=0;
int rint=0;
int rdens=0;
```

```
  char* datafile = argv[2];
  char* rowfile = argv[3];
  char* colfile = argv[4];

  for(long long int line_num=0; line_num<num_samples; line_num++){
    for(long long int col_num=0; col_num<num_samples; col_num++){
      if(line_num>=dimension && j<clusterNum){
        prevDimension=dimension;
        dimension=clusterDim[j];
        range = clusterD[j];
        orange = overlapP[j];
        j++;
        printf("line: %lld, density: %d, overlap: %d, old dimension: %d, new dimension: %d\n",
line_num, range, orange, prevDimension, dimension);
      }
      rint=rand() % (randrange);
      rdens=rand() % 100;
      if(dimension>line_num && dimension>col_num && prevDimension<=line_num &&
prevDimension<=col_num &&
        col_num>=line_num && density[j-1]>0 && rdens<range){
        if(col_num==line_num){
            if(sparse){
              Q[nz] = 1.000000;
              x[nz] = line_num;
              y[nz] = col_num;
              nz++;
            }
            else{
              Q[line_num * num_samples + col_num] = 1.000000;
            }
            density[j-1]--;
        }
        else{
            if(sparse){
              Q[nz] = rint/1000000.0;
              x[nz] = line_num;
              y[nz] = col_num;
              nz++;
              Q[nz] = rint/1000000.0;
```

```
            x[nz] = col_num;
            y[nz] = line_num;
            nz++;
          }
          else{
            Q[line_num * num_samples + col_num] = rint/1000000.0;
            Q[col_num * num_samples + line_num] = rint/1000000.0;
          }
          density[j-1]--;
          density[j-1]--;
      }
    }
    else if(overlapArea[j-1]>0 && dimension>line_num && col_num>=dimension &&
col_num<clusterDim[j] && rdens<orange && col_num>=line_num){
      if(sparse){
        Q[nz] = rint/1000000.0;
        x[nz] = line_num;
        y[nz] = col_num;
        nz++;
        Q[nz] = rint/1000000.0;
        x[nz] = col_num;
        y[nz] = line_num;
        nz++;
      }
      else{
        Q[line_num * num_samples + col_num] = rint/1000000.0;
        Q[col_num * num_samples + line_num] = rint/1000000.0;
      }
      overlapArea[j-1]--;
    }
    else if(j==1 && overlapArea[clusterNum-1]>0 && dimension>line_num &&
col_num>=clusterDim[clusterNum-2] && col_num<clusterDim[clusterNum-1] &&
rdens<overlapP[clusterNum-1] && col_num>=line_num){
      if(sparse){
        Q[nz] = rint/1000000.0;
        x[nz] = line_num;
        y[nz] = col_num;
        nz++;
        Q[nz] = rint/1000000.0;
        x[nz] = col_num;
```

```
      y[nz] = line_num;
      nz++;
    }
    else{
      Q[line_num * num_samples + col_num] = rint/1000000.0;
      Q[col_num * num_samples + line_num] = rint/1000000.0;
    }
    overlapArea[clusterNum-1]--;
  }
  else if(randElements>0 && rdens<randomP && clusterDim[j]<=col_num &&
col_num>=line_num){
    if(col_num==line_num){
      if(sparse){
        Q[nz] = 1.000000;
        x[nz] = line_num;
        y[nz] = col_num;
        nz++;
      }
      else{
        Q[line_num * num_samples + col_num] = 1.000000;
      }
      randElements--;
    }
    else{
      if(sparse){
        Q[nz] = rint/1000000.0;
        x[nz] = line_num;
        y[nz] = col_num;
        nz++;
        Q[nz] = rint/1000000.0;
        x[nz] = col_num;
        y[nz] = line_num;
        nz++;
      }
      else{
        Q[line_num * num_samples + col_num] = rint/1000000.0;
        Q[col_num * num_samples + line_num] = rint/1000000.0;
      }
      randElements--;
    }
```

```
    }
  }
}

if(sparse){
  FILE* file = fopen(datafile, "w");
  FILE* rfile = fopen(rowfile, "w");
  FILE* cfile = fopen(colfile, "w");
  if (file == NULL) {
    exit(EXIT_FAILURE);
  }
  if (rfile == NULL) {
    exit(EXIT_FAILURE);
  }
  if (cfile == NULL) {
    exit(EXIT_FAILURE);
  }
  for(long long int k=0; k<nz; k++){
    fprintf(file, "%f\n", Q[k]);
    fprintf(rfile, "%d\n", x[k]);
    fprintf(cfile, "%d\n", y[k]);
  }

  free(Q);
  free(x);
  free(y);
  fclose(file);
  fclose(rfile);
  fclose(cfile);
}
else{
  FILE* file = fopen(datafile, "w");
  if (file == NULL) {
    exit(EXIT_FAILURE);
  }
  printf("writing to file\n");
  for(long long int k=0; k<(num_samples*num_samples); k++){
    if(((k+1)%num_samples)==0){
      fprintf(file, "%f\n", Q[k]);
    }
```

```
      else{
        fprintf(file, "%f,", Q[k]);
      }
    }
    free(Q);
    fclose(file);
  }
  t1 = clock();
  printf("program took %f seconds to execute \n",
           (double) (t1 - t0) / CLOCKS_PER_SEC);
  return 0;
}
```

neurons_engine.c

```
#include <stdlib.h>
#include <stdio.h>
#include "//Users/jaesangha/Downloads/spblas_0_8/blas.h" // external library(blas). ADJUST
TO YOUR FILE LOCATION
#include <string.h>
#include <time.h>
#include <cblas.h>
#include <stdbool.h>

#define VERBOSE false
#define TAU 1
#define EPSILON 0.001
#define REGULARIZATION_FACTOR 0.8
#define ADDITIVE_FACTOR 1100

void validate_cmdline_args(int argc, int nargs, const char* msg) {
    if (argc < nargs) {
        printf("%s", msg);
        exit(EXIT_FAILURE);
    }
}

void validate_file(const char* in, const char* msg) {
    FILE* file = fopen(in, "r");
    if (file == NULL) {
        printf("%s", msg);
```

```
        exit(EXIT_FAILURE);
    }
    fclose(file);
}

void detect_spikes(int m, int n, double spike_value, double d[m * n],
                   double result[m * n]) {
    int i;
    int j;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            if (d[i * n + j] == spike_value) // spike
            {
                result[i * n + j] = 1;
            } else {
                result[i * n + j] = 0;
            }
        }
    }
}

void print_spikes(int m, int n, int iteration, double spikes[m * n],
                  const char* filename, const char* msg) {
    FILE* file = fopen(filename, "a");
    if (file == NULL) {
        printf("%s", msg);
        exit(EXIT_FAILURE);
    }
    int i;
    int j;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            if (spikes[i * n + j] == 1) {
                fprintf(file, "%d,%d\n", i, iteration);
            }
        }
    }
    fclose(file);
}
```

```
void print_margins(int m, int n, double margins[m * n], double labels[m * n],
                  const char* filename, const char* msg) {
    FILE* file = fopen(filename, "a");
    if (file == NULL) {
        printf("%s", msg);
        exit(EXIT_FAILURE);
    }
    int i;
    int j;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            if (labels[i * n + j] == 1) {
                fprintf(file, "%i,%f\n", i, margins[i * n + j]);
            }
        }
    }
    fclose(file);
}

int main(int argc, char* argv[]) {
    clock_t t0, t1;
    t0 = clock();
    validate_cmdline_args(argc, 7,
                        "Usage: ./neurons_engine <sparse> <dataset-file-name> <y-data-file-name>
<line-ind-file-name> <col-ind-file-name> <num-data-samples> <spike-event-output-file-name>
<num-iterations>\n");
    bool sparse_mat = atoi(argv[1]);
    char* dataset_file_name = argv[2];
    char* Y_file_name = argv[3];
    char* line_ind_file_name = argv[4];
    char* col_ind_file_name = argv[5];
    validate_file(dataset_file_name, "Dataset file name must be valid\n");
    validate_file(Y_file_name, "Y file name must be valid\n");
    validate_file(line_ind_file_name, "line ind file name must be valid\n");
    validate_file(col_ind_file_name, "col ind file name must be valid\n");
    long long int num_samples = atoi(argv[6]);
    char* spike_event_output_file_name = argv[7];
    int num_iterations = atoi(argv[8]);

    // Construct Q,Y matrix from input file and determine the sparsity
```

```
double *Y = (double *) malloc(num_samples * 2 * sizeof(double));

char line[10];
long long int line_num = 0;
long long int ind_num = 0;
long long int size_newQ = 0;
if(sparse_mat){
  size_newQ = num_samples * num_samples * 0.01;
}
else{
  size_newQ = num_samples * num_samples;
}
double *newQ = (double*) malloc(size_newQ * sizeof(double));
int *xind = (int*) malloc(size_newQ * sizeof(int));
int *yind = (int*) malloc(size_newQ * sizeof(int));

FILE* inputY = fopen(Y_file_name, "r");
while (fgets(line, sizeof(line), inputY)) {
    long long int col_num = 0;
    char* token;
    token = strtok(line, ",");
    while (token != NULL) {
        Y[line_num * 2 + col_num] = atof(token);
        //          printf("%d\n",(int)Y[line_num * 2 + col_num]);
        token = strtok(NULL, ",");
        col_num++;
    }
    line_num++;
}
fclose(inputY);
if(sparse_mat){
  long long int x_num=0;
  FILE* inputLineind = fopen(line_ind_file_name, "r");
  while (fgets(line, sizeof(line), inputLineind)) {
      char* token;
      token = strtok(line, ",");
      while (token != NULL) {
          xind[x_num] = atof(token);
          token = strtok(NULL, ",");
          x_num++;
```

```
    }
  }
  fclose(inputLineind);
  long long int y_num=0;
  FILE* inputColind = fopen(col_ind_file_name, "r");
  while (fgets(line, sizeof(line), inputColind)) {
    char* token;
    token = strtok(line, ",");
    while (token != NULL) {
      yind[y_num] = atof(token);
      token = strtok(NULL, ",");
      y_num++;
    }
  }
  fclose(inputColind);
  FILE* file = fopen(dataset_file_name, "r");
  while (fgets(line, sizeof(line), file)) {
    char* token;
    token = strtok(line, ",");
    while (token != NULL) {
      newQ[ind_num] = atof(token);
      ind_num++;
      token = strtok(NULL, ",");
    }
  }
  fclose(file);
}
else{
  line_num = 0;
  FILE* file = fopen(dataset_file_name, "r");
  char line[num_samples * 9 + 1];
  while (fgets(line, sizeof(line), file)) {
    int col_num = 0;
    char* token;
    token = strtok(line, ",");
    while (token != NULL) {
        newQ[line_num * num_samples + col_num] = atof(token);
        token = strtok(NULL, ",");
        col_num++;
    }
```

```
      line_num++;
    }
    fclose(file);
  }


double *P = (double *) malloc(num_samples * 2 * sizeof(double));
memset(P, 0, num_samples * 2 * sizeof(P[0]));
double *avgP = (double *) malloc(num_samples * 2 * sizeof(double));
memset(avgP, 0, num_samples * 2 * sizeof(avgP[0]));
int avg_num = 100;
if (avg_num > num_iterations) { // take average out of 100 if iteration is over 100
   avg_num = num_iterations;
}
double *ones = (double *) malloc(num_samples * 2 * sizeof(double));
for (int k = 0; k < num_samples * 2; k++) {
   ones[k] = 1;
}
long long int nz = ind_num;
cblas_daxpy(num_samples * 2, 1.0 / num_samples, ones, 1, P, 1); //initialize P
long long int num_neurons = num_samples;
blas_sparse_matrix A, minusA;
if (sparse_mat) {
   // Construct sparse matrix representation
   A = BLAS_duscr_begin(num_samples, num_samples);
   printf("sparse matrix build\n");
   for (int j = 0; j < nz; j++) {
      BLAS_duscr_insert_entry(A, newQ[j], xind[j], yind[j]);
   }
   BLAS_duscr_end(A);
   printf("symmetric?: %d\n", BLAS_usgp(A, blas_symmetric));
}
double *b = (double *) malloc(num_neurons * 2 * sizeof(double));
memset(b, 0, num_samples * 2 * sizeof(b[0]));
if (sparse_mat) {
   for (int k = 0; k < 2; k++) {
      BLAS_dusmv(blas_no_trans, 1.0, A, Y + k, 2, b + k, 2); //SPARSE COMPUTATION
   }
} else {
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, num_neurons, 2,
```

```
        num_neurons, 1.0, newQ, num_neurons, Y, 2, 0.0, b, 2);
}
// Y = Q^(-1)*b
printf("#: %lld\nsize: %lld\n", nz, size_newQ);
double *minusQ = (double *) malloc(num_samples * num_samples * sizeof(double));
if (sparse_mat) {
  cblas_daxpy(nz, -2, newQ, 1, newQ, 1); //minusQ = -Q
  minusA = BLAS_duscr_begin(num_samples, num_samples);
  for (int j = 0; j < nz; j++) {
    BLAS_duscr_insert_entry(minusA, newQ[j], xind[j], yind[j]);
  }
  BLAS_duscr_end(minusA);
} else {
  cblas_daxpy(num_neurons * num_neurons, -1, newQ, 1, minusQ, 1); //minusQ = -Q
}
if(sparse_mat){
 free(newQ);
 free(minusQ);
}
free(xind);
free(yind);
// Update loop is here (detect the spikes/generate spike output file)
for (int i = 0; i < num_iterations; i++) {
  int ii;
  double W = 10;
  double *d = (double *) malloc(num_neurons * 2 * sizeof(double));
  for (ii = 0; ii < num_neurons * 2; ii++) {
    if ((0 <= P[ii]) && (P[ii] < (0.5 - EPSILON))) {
      d[ii] = -1;
    } else if (((0.5 - EPSILON) <= P[ii])
          && (P[ii] <= (0.5 + EPSILON))) {
      d[ii] = W * (((P[ii] - 0.5) > 0) - ((P[ii] - 0.5) < 0));
    } else if (((0.5 + EPSILON) < P[ii]) && (P[ii] <= 1)) {
      d[ii] = 1;
    } else {
      printf("line: %lld, col: %lld; Invalid P value: %f\n",
          ii / num_neurons + 1, ii % num_neurons, P[ii]);
      exit(EXIT_FAILURE);
    }
  }
```

```
double *Pest = (double *) malloc(num_neurons * 2 * sizeof(double));
cblas_dcopy(num_neurons * 2, P, 1, Pest, 1); //Pest = P
double *newP = (double *) malloc(num_neurons * 2 * sizeof(double));
cblas_dcopy(num_neurons * 2, b, 1, newP, 1); //newP = b
double *intermediate1 = (double *) malloc(num_neurons * 2 * sizeof(double));
memset(intermediate1, 0, num_neurons * 2 * sizeof(intermediate1[0]));
if (sparse_mat) {
   for (int k = 0; k < 2; k++) {
      BLAS_dusmv(blas_no_trans, 1.0, minusA, Pest + k, 2,
              intermediate1 + k, 2); //SPARSE COMPUTATION
   }
} else {
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, num_neurons,
             2, num_neurons, 1.0, minusQ, num_neurons, Pest, 2, 0.0,
             intermediate1, 2); //intermediate1 = -Q*Pest
}
cblas_daxpy(num_neurons * 2, -REGULARIZATION_FACTOR, d, 1, newP, 1); //newP = b
-rd
cblas_daxpy(num_neurons * 2, ADDITIVE_FACTOR, ones, 1, newP, 1); //newP = b-
rd+ADDITIVE_FACTOR
cblas_daxpy(num_neurons * 2, 1, intermediate1, 1, newP, 1); //newP = -Q*Pest+ b-
rd+ADDITIVE_fACTOR

double *Z = (double *) malloc(num_neurons * 2 * sizeof(double));
cblas_dsbmv(CblasRowMajor, CblasUpper, num_neurons * 2, 0, 1.0, P, 1,
        newP, 1, 0.0, Z, 1); //Z=P.*newP
double *sumZ = (double *) malloc(num_neurons * sizeof(double));
double ones_2_1[] = { 1, 1 };
cblas_dgemv(CblasRowMajor, CblasNoTrans, num_neurons, 2, 1.0, Z, 2,
        ones_2_1, 1, 0.0, sumZ, 1);
double *normZ = (double *) malloc(num_neurons * 2 * sizeof(double));
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, num_neurons, 2, 1,
        1.0, sumZ, 1, ones_2_1, 1, 0.0, normZ, 2); //normZ = sumZ*ones
//normZ = Z ./ sumZ
for (int j = 0; j < num_neurons * 2; j++) {
   normZ[j] = TAU * (Z[j] / normZ[j]);
}
cblas_daxpy(num_neurons * 2, 1 - TAU, P, 1, normZ, 1); //normZ = (1-tau)P + normZ
cblas_dcopy(num_neurons * 2, normZ, 1, P, 1); //P = normZ
if (num_iterations > avg_num && i >= (num_iterations - avg_num)) {
```

```
            cblas_daxpy(num_neurons * 2, 1, P, 1, avgP, 1); //avgP = P+avgP
        } else if (num_iterations == avg_num) {
            cblas_daxpy(num_neurons * 2, 1, P, 1, avgP, 1); //avgP = P+avgP
        }
        double *spikes = (double *) malloc(num_neurons * 2 * sizeof(double));
        detect_spikes(num_neurons, 2, W, d, spikes);
        print_spikes(num_neurons, 2, i, spikes, spike_event_output_file_name,
                "Enter a valid output filename\n");
        free(spikes);
        free(d);
        free(Pest);
        free(newP);
        free(intermediate1);
        free(Z);
        free(sumZ);
        free(normZ);
    }
    free(avgP);
    free(P);
    free(ones);
    free(b);
    free(Y);
    t1 = clock();
    printf("program took %f seconds to execute \n",
        (double) (t1 - t0) / CLOCKS_PER_SEC);
    return 0;
}
```

gui.py

```
import sys
import time
import math
import os
import tkMessageBox
from Tkinter import *
import ttk
import matplotlib
from numpy import arange, sin, pi
matplotlib.use('TkAgg')
from matplotlib.backends.backend_tkagg import \
```

```
    FigureCanvasTkAgg,NavigationToolbar2TkAgg
from matplotlib.figure import Figure
import matplotlib.pyplot as plt

root = Tk()

nb = ttk.Notebook(root)

# adding Frames as pages for the ttk.Notebook
# first page, which would get widgets gridded into it
page1 = ttk.Frame(nb)

# second page
page2 = ttk.Frame(nb)

nb.add(page1, text='Coupling Matrix')
nb.add(page2, text='Spike Plot')

nb.pack(expand=1, fill="both")

canvasFrame = Frame(page1)
canvasFrame.pack(side=RIGHT)
qCanvas = Canvas(canvasFrame, bg='gray', width='400', height='400')
qCanvas.pack(side=TOP)

rasterCanvas = Canvas(page2, bg='black', width='600', height='200')
rasterCanvas.pack()
fig2=plt.Figure(figsize=(5,3.5))
ax2 = fig2.add_subplot(111)
canvas_raster=FigureCanvasTkAgg(fig2,rasterCanvas)
canvas_raster.get_tk_widget().pack()

class MyGrid(Frame):
    def __init__(self, master = None):
        self.build2()
        self.build()
    def build2(self):
        Frame.__init__(self, page2)
        self.pack()
        self.e8 = DoubleVar()
```

```
    self.e11 = DoubleVar()
    self.e12 = IntVar()
    self.e13 = IntVar()
    self.e14 = IntVar()
    self.e15 = IntVar()
    Label(self, text = "neuron id range").grid(row=5, column=0)
    Entry(self,textvariable=self.e12,width=10).grid(row = 5, column = 1)
    Entry(self,textvariable=self.e13,width=10).grid(row = 5, column = 2)
    Label(self, text = "time(iteration) range").grid()
    Entry(self,textvariable=self.e14,width=10).grid(row = 6, column = 1)
    Entry(self,textvariable=self.e15,width=10).grid(row = 6, column = 2)
    Button(self, text = "Generate Plot",command=self.genPlot).grid(columnspan=3)

def build(self):
    Frame.__init__(self, page1)
    self.pack()
    Label(self, text = "Coupling Matrix").grid()
    Label(self, text = "# neurons").grid()
    Label(self, text = "Matrix type").grid()
    Label(self, text = "width(%)").grid(row=3, column=1)
    Label(self, text = "density(%)").grid(row=3, column=2)
    Label(self, text = "1st cluster").grid() #row 3
    Label(self, text = "2nd cluster").grid()
    Label(self, text = "3rd cluster").grid()
    Label(self, text = "4th cluster").grid()
    Label(self, text = "5th cluster").grid()
    Label(self, text = "6th cluster").grid()
    Label(self, text = "7th cluster").grid()
    Label(self, text = "8th cluster").grid()
    Label(self, text = "9th cluster").grid()
    Label(self, text = "10th cluster").grid() #row 12
    Label(self, text = "overlap?").grid()
    Label(self, text = "data filename").grid()
    Label(self, text = "row filename(only for sparse)").grid()
    Label(self, text = "col filename(only for sparse)").grid()
    self.e1 = StringVar()
    optionList1 = ('1000', '10000', '100000', '1000000')
    self.e1.set(optionList1[0])
    self.e2 = StringVar()
    optionList2 = ('sparse', 'dense')
```

```
optionList3 = ('sparse')
self.e2.set(optionList2[0])
self.c1 = IntVar()
self.c1.set(50)
self.c2 = IntVar()
self.c3 = IntVar()
self.c4 = IntVar()
self.c5 = IntVar()
self.c6 = IntVar()
self.c7 = IntVar()
self.c8 = IntVar()
self.c9 = IntVar()
self.c10 = IntVar()
self.d1 = IntVar()
self.d1.set(50)
self.d2 = IntVar()
self.d3 = IntVar()
self.d4 = IntVar()
self.d5 = IntVar()
self.d6 = IntVar()
self.d7 = IntVar()
self.d8 = IntVar()
self.d9 = IntVar()
self.d10 = IntVar()
self.c1_o = IntVar()
self.c2_o = IntVar()
self.c3_o = IntVar()
self.c4_o = IntVar()
self.c5_o = IntVar()
self.c6_o = IntVar()
self.c7_o = IntVar()
self.c8_o = IntVar()
self.c9_o = IntVar()
self.c10_o = IntVar()
self.e5 = StringVar()
self.e6 = StringVar()
self.e7 = StringVar()
self.e9 = StringVar()
self.e9.set("0")
self.e16 = IntVar()
```

```
        self.e16.set("1000")
        self.rowF = StringVar()
        self.colF = StringVar()

        OptionMenu(self, self.e1, *optionList1).grid(row=1,column=1)
        OptionMenu(self, self.e2, *optionList2).grid(row=2,column=1)
        Entry(self,textvariable=self.c1).grid(row = 4, column = 1)
        Entry(self,textvariable=self.c2).grid(row = 5, column = 1)
        Entry(self,textvariable=self.c3).grid(row = 6, column = 1)
        Entry(self,textvariable=self.c4).grid(row = 7, column = 1)
        Entry(self,textvariable=self.c5).grid(row = 8, column = 1)
        Entry(self,textvariable=self.c6).grid(row = 9, column = 1)
        Entry(self,textvariable=self.c7).grid(row = 10, column = 1)
        Entry(self,textvariable=self.c8).grid(row = 11, column = 1)
        Entry(self,textvariable=self.c9).grid(row = 12, column = 1)
        Entry(self,textvariable=self.c10).grid(row = 13, column = 1)
        Entry(self,textvariable=self.d1).grid(row = 4, column = 2)
        Entry(self,textvariable=self.d2).grid(row = 5, column = 2)
        Entry(self,textvariable=self.d3).grid(row = 6, column = 2)
        Entry(self,textvariable=self.d4).grid(row = 7, column = 2)
        Entry(self,textvariable=self.d5).grid(row = 8, column = 2)
        Entry(self,textvariable=self.d6).grid(row = 9, column = 2)
        Entry(self,textvariable=self.d7).grid(row = 10, column = 2)
        Entry(self,textvariable=self.d8).grid(row = 11, column = 2)
        Entry(self,textvariable=self.d9).grid(row = 12, column = 2)
        Entry(self,textvariable=self.d10).grid(row = 13, column = 2)

        Checkbutton(self,variable=self.e9,command=self.cb).grid(row = 14, column = 1)
        Entry(self,textvariable=self.e5).grid(row = 15, column = 1)
        Entry(self,textvariable=self.rowF).grid(row = 16, column = 1)
        Entry(self,textvariable=self.colF).grid(row = 17, column = 1)
        Button(self, text = "Plot Interconnection Matrix",
command=self.plotMat).grid(row=18,column=0)
        Button(self, text = "Generate Interconnection Matrix",
command=self.genMat).grid(row=18,column=1)
        Label(self, text = "plot(spike) filename").grid(row=19,column=0)
        Entry(self,textvariable=self.e6).grid(row = 19, column = 1)
        Label(self, text = "# iterations").grid(row=20,column=0)
        Entry(self,textvariable=self.e16).grid(row = 20, column = 1)
        Button(self, text = "RUN", command=self.run).grid(columnspan=2)
```

```
def genMat(self):
    matrixType = self.e2.get()
    numNeurons= int(self.e1.get())
    rFile= self.rowF.get()+'.txt'
    cFile= self.colF.get()+'.txt'
    overlap=self.e9.get()
    dataFile= self.e5.get()+'.txt'
    clusters=[]
    clusters.append(self.c1.get())
    clusters.append(self.c2.get())
    clusters.append(self.c3.get())
    clusters.append(self.c4.get())
    clusters.append(self.c5.get())
    clusters.append(self.c6.get())
    clusters.append(self.c7.get())
    clusters.append(self.c8.get())
    clusters.append(self.c9.get())
    clusters.append(self.c10.get())
    density=[]
    density.append(self.d1.get())
    density.append(self.d2.get())
    density.append(self.d3.get())
    density.append(self.d4.get())
    density.append(self.d5.get())
    density.append(self.d6.get())
    density.append(self.d7.get())
    density.append(self.d8.get())
    density.append(self.d9.get())
    density.append(self.d10.get())
    overlap=[]
    overlap.append(self.c1_o.get())
    overlap.append(self.c2_o.get())
    overlap.append(self.c3_o.get())
    overlap.append(self.c4_o.get())
    overlap.append(self.c5_o.get())
    overlap.append(self.c6_o.get())
    overlap.append(self.c7_o.get())
    overlap.append(self.c8_o.get())
    overlap.append(self.c9_o.get())
    overlap.append(self.c10_o.get())
```

```
        totalWidth=0
        for c in clusters:
            if c>0:
                totalWidth+=c
                if totalWidth>100:
                    tkMessageBox.showerror("Error","cluster width exceeds 100% limit")
                    return
            else:
                break;
        for d in density:
            if d>100 or d<0:
                tkMessageBox.showerror("Error","cluster density has to be within 0 ~ 100 %")
                return
        for o in overlap:
            if o>100 or o<0:
                tkMessageBox.showerror("Error","overlap % has to be within 0 ~ 100 %")
                return
        os.system('gcc -g ../neurons_gendata.c -Wall -Werror -o ../neurons_gendata')
        if matrixType=="dense":
            if numNeurons>=100000:
                tkMessageBox.showerror("Error","If the # of neurons are greater than or equal to
100000, matrix type has to be sparse")
                return
            os.system('../../neurons_gendata 0 ../Dataset/'+dataFile+' ../Dataset/'+rFile+'
../Dataset/'+cFile+' '+str(numNeurons)+' '+str(self.numCluster)
            +' '+str(clusters[0])+' '+str(clusters[1])+' '+str(clusters[2])+' '+str(clusters[3])+'
'+str(clusters[4])+' '+str(clusters[5])+' '+str(clusters[6])+' '+str(clusters[7])
            +' '+str(clusters[8])+' '+str(clusters[9])
            +' '+str(density[0])+' '+str(density[1])+' '+str(density[2])+' '+str(density[3])+'
'+str(density[4])+' '+str(density[5])+' '+str(density[6])+' '+str(density[7])
            +' '+str(density[8])+' '+str(density[9])
            +' '+str(overlap[0])+' '+str(overlap[1])+' '+str(overlap[2])+' '+str(overlap[3])+'
'+str(overlap[4])+' '+str(overlap[5])+' '+str(overlap[6])+' '+str(overlap[7])
            +' '+str(overlap[8])+' '+str(overlap[9]))
            tkMessageBox.showinfo("Generate the coupling matrix","Execution done")
            return
        else:
            os.system('../../neurons_gendata 1 ../Dataset/'+dataFile+' ../Dataset/'+rFile+'
../Dataset/'+cFile+' '+str(numNeurons)+' '+str(self.numCluster)
```

```
            +' '+str(clusters[0])+' '+str(clusters[1])+' '+str(clusters[2])+' '+str(clusters[3])+'
'+str(clusters[4])+' '+str(clusters[5])+' '+str(clusters[6])+' '+str(clusters[7])
            +' '+str(clusters[8])+' '+str(clusters[9])
            +' '+str(density[0])+' '+str(density[1])+' '+str(density[2])+' '+str(density[3])+'
'+str(density[4])+' '+str(density[5])+' '+str(density[6])+' '+str(density[7])
            +' '+str(density[8])+' '+str(density[9])
            +' '+str(overlap[0])+' '+str(overlap[1])+' '+str(overlap[2])+' '+str(overlap[3])+'
'+str(overlap[4])+' '+str(overlap[5])+' '+str(overlap[6])+' '+str(overlap[7])
            +' '+str(overlap[8])+' '+str(overlap[9]))
            tkMessageBox.showinfo("Generate the coupling matrix","Execution done")
            return
    def cb(self):
        if int(self.e9.get())==1:
            Label(self, text = "overlap % with next cluster").grid(row=3, column=3)
            self.o1 = Entry(self,textvariable=self.c1_o)
            self.o1.grid(row = 4, column = 3)
            self.o2 = Entry(self,textvariable=self.c2_o)
            self.o2.grid(row = 5, column = 3)
            self.o3 = Entry(self,textvariable=self.c3_o)
            self.o3.grid(row = 6, column = 3)
            self.o4 = Entry(self,textvariable=self.c4_o)
            self.o4.grid(row = 7, column = 3)
            self.o5 = Entry(self,textvariable=self.c5_o)
            self.o5.grid(row = 8, column = 3)
            self.o6 = Entry(self,textvariable=self.c6_o)
            self.o6.grid(row = 9, column = 3)
            self.o7 = Entry(self,textvariable=self.c7_o)
            self.o7.grid(row = 10, column = 3)
            self.o8 = Entry(self,textvariable=self.c8_o)
            self.o8.grid(row = 11, column = 3)
            self.o9 = Entry(self,textvariable=self.c9_o)
            self.o9.grid(row = 12, column = 3)
            self.o10 = Entry(self,textvariable=self.c10_o)
            self.o10.grid(row = 13, column = 3)
        else:
            self.o1.destroy()
            self.o2.destroy()
            self.o3.destroy()
            self.o4.destroy()
            self.o5.destroy()
```

```
            self.o6.destroy()
            self.o7.destroy()
            self.o8.destroy()
            self.o9.destroy()
            self.o10.destroy()
    def run(self):
        matrixType = self.e2.get()
        qFile= self.e5.get()+'.txt'
        rFile= self.rowF.get()+'.txt'
        cFile= self.colF.get()+'.txt'
        plotFile= self.e6.get()+'.txt'
        numNeurons= int(self.e1.get())
        numIteration=self.e16.get()
        if numIteration<=0:
            tkMessageBox.showerror("Error","Iteration can not be less than or equal to 0")
            return
        spikeFunc=""
        os.system('gcc -g -I/Users/jaesangha/Downloads/spblas_0_8/*.h
/Users/jaesangha/Downloads/spblas_0_8/*.c ../neurons_engine.c -o ../neurons_engine -Wall -
Werror -lcblas -lblas')
        if matrixType=="dense":
            if numNeurons>=100000:
                tkMessageBox.showerror("Error","If the # of neurons are greater than or equal to
100000, matrix type has to be sparse")
                return
            os.system('../../neurons_engine 0 ../Dataset/'+qFile+'
../Dataset/y_'+str(numNeurons)+'.txt ../Dataset/'+rFile+' ../Dataset/'+cFile+'
'+str(numNeurons)+' ../'+plotFile+' '+str(numIteration))
            tkMessageBox.showinfo("Run the simulation","Execution done")
            return
        else:
            os.system('../../neurons_engine 1 ../Dataset/'+qFile+'
../Dataset/y_'+str(numNeurons)+'.txt ../Dataset/'+rFile+' ../Dataset/'+cFile+'
'+str(numNeurons)+' ../'+plotFile+' '+str(numIteration))
            tkMessageBox.showinfo("Run the simulation","Execution done")
            return
    def genSpikeFunc(self):
        w= self.e8.get()
        h= self.e11.get()
        x = [0, 0.5-w, 0.5-w, 0.5, 0.5, 0.5+w, 0.5+w, 1]
```

```
    y = [-1, -1, -h, -h, h, h, 1, 1]
    x3 = [0, 0.5, 0.5, 0.5+w, 0.5+w, 1]
    y3 = [-1, -1, h,     h, 0, 0]
    ax1.clear()
    ax1.plot(x, y)
    ax1.grid(True)
    ax1.set_xlim((0, 1))
    ax1.set_title('Spike Function')
    ax3.clear()
    ax3.plot(x3, y3)
    ax3.grid(True)
    ax3.set_xlim((0, 1))
    ax3.set_title('Spike Function')
    canvas.show()
  def genPlot(self):
    print "PLOTTING"
    neuron_from=self.e12.get()
    neuron_to=self.e13.get()
    time_from=float(self.e14.get())
    time_to=float(self.e15.get())
    if neuron_from<0 or time_from<0 or neuron_to<0 or time_to <0:
      tkMessageBox.showerror("Error","Neuron id/Time range can not contain negative value")
      return
    num_neurons = int(self.e1.get())
    it = int(self.e16.get())
    if neuron_from>num_neurons or time_from>it or neuron_to>num_neurons or time_to>it:
      tkMessageBox.showerror("Error","Neuron id/Time value exceeds range")
      return

    if neuron_from==0 and neuron_to==0:
      neuron_to=num_neurons
    if time_from==0 and time_to==0:
      time_to=int(self.e16.get())
    in_="../" + self.e6.get() +".txt"
    print(os.path.exists(in_))
    pullData = open(in_, 'r').read()
    dataArray = pullData.split('\n')
    time_ar = []
    n_id_ar = []
    row=0
```

```
    for eachLine in dataArray:
        if len(eachLine) > 1:
            n, t = eachLine.split(',')
            if float(t)<=time_to and float(t)>=time_from:
                time_ar.append(float(t))
            else:
                time_ar.append(0.0)
            if int(n)<=neuron_to and int(n)>=neuron_from:
                n_id_ar.append(int(n))
            else:
                n_id_ar.append(0)
    ax2.clear()
    ax2.plot(time_ar, n_id_ar, 'k|', markeredgewidth=1.0)
    ax2.set_ylim(ax2.get_ylim()[::-1])
    ax2.set_title('Neuron Spiking Events')
    ax2.set_xlabel('Time')
    ax2.set_ylabel('Neuron ID')
    ax2.set_ylim([neuron_to,neuron_from])
    ax2.set_xlim([time_from,time_to])
    canvas_raster.show()
def plotMat(self):
    qCanvas.delete("all")
    numNeurons= int(self.e1.get())
    matrixType= self.e2.get()
    print matrixType
    clusters=[]
    clusters.append(self.c1.get())
    clusters.append(self.c2.get())
    clusters.append(self.c3.get())
    clusters.append(self.c4.get())
    clusters.append(self.c5.get())
    clusters.append(self.c6.get())
    clusters.append(self.c7.get())
    clusters.append(self.c8.get())
    clusters.append(self.c9.get())
    clusters.append(self.c10.get())
    density=[]
    density.append(self.d1.get())
    density.append(self.d2.get())
    density.append(self.d3.get())
```

```
density.append(self.d4.get())
density.append(self.d5.get())
density.append(self.d6.get())
density.append(self.d7.get())
density.append(self.d8.get())
density.append(self.d9.get())
density.append(self.d10.get())
overlaps=[]
overlaps.append(self.c1_o.get())
overlaps.append(self.c2_o.get())
overlaps.append(self.c3_o.get())
overlaps.append(self.c4_o.get())
overlaps.append(self.c5_o.get())
overlaps.append(self.c6_o.get())
overlaps.append(self.c7_o.get())
overlaps.append(self.c8_o.get())
overlaps.append(self.c9_o.get())
overlaps.append(self.c10_o.get())
dataFile= self.e5.get()
steps=[]
self.numCluster=0
totalWidth = 0
if matrixType=="sparse":
   sparsity=400*400*0.01
   for c in clusters:
      if c>0:
         totalWidth+=c
         if totalWidth>100:
            tkMessageBox.showerror("Error","cluster width exceeds 100% limit")
            return
         steps.append(math.sqrt(sparsity*c/100))
         self.numCluster+=1
      elif c<0:
            tkMessageBox.showerror("Error","cluster width has to be within 1 ~ 100 %")
            return
      else:
         break;
else:
   for c in clusters:
      if c>0:
```

```
            totalWidth+=c
            if totalWidth>100:
                tkMessageBox.showerror("Error","cluster width exceeds 100% limit")
                return
            steps.append(400*c/100)
            self.numCluster+=1
        elif c<0:
                tkMessageBox.showerror("Error","cluster width has to be within 1 ~ 100 %")
                return
        else:
            break;
for d in density:
    if d>100 or d<0:
        tkMessageBox.showerror("Error","cluster density has to be within 1 ~ 100 %")
        return
for o in overlaps:
    if o>100 or o<0:
        tkMessageBox.showerror("Error","overlap % has to be within 0 ~ 100 %")
        return
x0=0
x1=0
y0=0
y1=0
o=0;
for s in steps:
    x0=x1
    y0=y1
    x1+=s
    y1+=s
    id = qCanvas.create_rectangle ( x0, y0, x1, y1, fill='black')
    text=str(density[o])+'%'
    id = qCanvas.create_text ( (x0+x1)/2, (y0+y1)/2, text=text, fill='white')
    if overlaps[o]>0:
        if o==self.numCluster-1:
            id = qCanvas.create_rectangle ( x0, 0, x1, steps[0], fill='')
            text=str(overlaps[o])+'%'
            id = qCanvas.create_text ( (x0+x1)/2, steps[0]/2, text=text, fill='white')
            id = qCanvas.create_rectangle ( 0, x0, steps[0], x1, fill='')
            text=str(overlaps[o])+'%'
            id = qCanvas.create_text ( steps[0]/2, (x0+x1)/2, text=text, fill='white')
```

```
            else:
                id = qCanvas.create_rectangle ( x1, y0, x1+steps[o+1], y1, fill='')
                text=str(overlaps[o])+'%'
                id = qCanvas.create_text ( (x1+x1+steps[o+1])/2, (y0+y1)/2, text=text, fill='white')
                id = qCanvas.create_rectangle ( y0, x1, y1, x1+steps[o+1], fill='')
                text=str(overlaps[o])+'%'
                id = qCanvas.create_text (  (y0+y1)/2, (x1+x1+steps[o+1])/2, text=text, fill='white')
        o+=1
if __name__ == "__main__":
    MyGrid().mainloop()
    MyGrid2().mainloop()
```

References

[1] Gangopadhyay, Ahana, and Shantanu Chakrabartty. "Spiking, Bursting, and Population Dynamics in a Network of Growth Transform Neurons." IEEE transactions on neural networks and learning systems (2017).

[2] Zenke, Friedemann, and Wulfram Gerstner. "Limits to high-speed simulations of spiking neural networks using general-purpose computers." Frontiers in neuroinformatics 8 (2014): 76.

[3] Vitay, Julien, Helge Ülo Dinkelbach, and Fred H. Hamker. "ANNarchy: a code generation approach to neural simulations on parallel hardware." Frontiers in neuroinformatics 9 (2015): 19.

[4] Soman, Sumit, and Manan Suri. "Recent trends in neuromorphic engineering." Big Data Analytics 1.1 (2016): 15.

[5] Yamazaki, Tadashi, and Jun Igarashi. "Realtime cerebellum: A large-scale spiking network model of the cerebellum that runs in realtime using a graphics processing unit." Neural Networks47 (2013): 103-111.

[6] Izhikevich, Eugene M. "Which model to use for cortical spiking neurons?." IEEE transactions on neural networks 15.5 (2004): 1063-1070.

[7] Goodman, Dan FM, and Romain Brette. "The brian simulator." Frontiers in neuroscience 3 (2009): 26.

[8] Hines, Michael L., and Nicholas T. Carnevale. "NEURON: a tool for neuroscientists." The neuroscientist 7.2 (2001): 123-135.

[9] Gewaltig, Marc-Oliver, Abigail Morrison, and Hans Ekkehard Plesser. "NEST by Example: An Introduction to the Neural Simulation Tool NEST Version 2.6. 0." (2013).

[10] Vreeken, Jilles. "Spiking neural networks, an introduction." (2003).

[11] Wikipedia contributors. "Neuroimaging." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 22 Apr. 2018. Web. 23 Apr. 2018.

[12] Beeman, David. "Hodgkin-huxley model." Encyclopedia of Computational Neuroscience (2013): 1-13.

[13] Orhan, Emin. "The leaky integrate-and-fire neuron model." no3 (2012): 1-6.

[14] Smart, John. "Preserving the Self for Later Emulation: What Brain Features Do We Need?" New Technology | Stock Discussion Forums, 30 Oct. 2012, www.siliconinvestor.com/readmsgs.aspx?subjectid=57958&msgnum=158&batchsize=10&batchtype=Next.

[15] Basegmez, Erdem. "The next generation neural networks: Deep learning and spiking neural networks." Advanced Seminar in Technical University of Munich. 2014.

[16] Duff, Iain S., Michael A. Heroux, and Roldan Pozo. "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum." ACM Transactions on Mathematical Software (TOMS) 28.2 (2002): 239-267.

[17] Blackford, L. Susan, et al. "An updated set of basic linear algebra subprograms (BLAS)." ACM Transactions on Mathematical Software 28.2 (2002): 135-15