Introduction to Spiking Network Simulation (a.k.a. `snsbook`)
— From a Single Neuron to a Human-Scale Whole Brain Model —

Tadashi Yamazaki
Graduate School of Informatics and Engineering
the University of Electro-Communications

Jun Igarashi
Center for Computational Science
RIKEN

January 5, 2022

Disclaimer

This is a machine-translated text of the following book:

> Tadashi Yamazaki and Jun Igarachi. *Introduction to Spiking Network Simulation* (published in Japanese). Morikita Publishing (2021).
> `https://www.morikita.co.jp/books/mid/085631`

The English version is generated semi-automatically by using DeepL Pro, so the translation is not perfect. However, the authors believe that it is worth of publishing in spite of the quality by the following reasons: (1) the text looks at least human-readable, (2) we could say it is better than never, and (3) in a sense, the body of the book is the code (`https://github.com/numericalbrainorg/snsbook/`) rather than the text.

The contents will be updated without any notifications, because we are still revising the contents slowly and continously. The published version should always be the latest version. Please do not forget to check the latest version.

Please send comments and suggestions to `snsbook-at-numericalbrain-dot-org`. They are warmly appreciated.

When you refer to the contents of this document, please cite our Japanese book shown above.

The Authors (Tadashi Yamazaki and Jun Igarashi)

Contents

Chapter 0

Opening

0.1 Foreword

TY: Foreword is given by Dr. Kenji Doya at OIST. This is exclusive for the Japanese version!!!

0.2 Preface

The brain is a huge and complex network of many neurons, called neurons, that are connected through structures called synapses. On the network, neurons exchange electric pulses called spikes to process information, and various brain functions appear as the function of the entire network. In order to understand this process, and to elucidate how functions are generated from the structure of the network, it is necessary to carefully examine how individual neurons fire spikes, how these spikes propagate through the network, and finally how the network as a whole behaves. We need to carefully examine how each individual spike fires and how that spike propagates through the network and ultimately how the network as a whole behaves.

However, it is very difficult to do so experimentally. However, it is very difficult to do so experimentally, because the brain contains a huge number of neurons and synapses, and it is impossible to measure each one of them simultaneously with sufficient resolution. On the other hand, computational simulations are possible, which allow us to take into account the interaction of all neurons, and to reproduce and predict neural activity.

This book is an introduction to spiking network simulation, which is used to simulate and predict the process from the firing of a spike by a single neuron to the creation of various functions by a network of neurons. It goes into the details of numerical and parallel computation that are not covered in conventional textbooks on computational neuroscience and neuroinformatics, and explicitly describes everything from neuron models to numerical methods at the algorithmic level. As a programming language, we will use the C language, which is suitable for such applications and supports OpenMP, MPI, and CUDA, all of which are necessary for parallel computing.

The title of this book is "Introduction to Spiking Network Simulation", because most readers will be new to spiking network simulation, and at the same time, we hope that it will be the first book that covers everything from general neuroscience to high-performance computing, in order to establish spiking network simulation / simulation neuroscience as an independent field from computational neuroscience and neuroinformatics. We hope that this book will be the first book that covers everything from general neuroscience to high-performance computing.

The authors each have lectures and student experiments at universities and other institutions, and in 2019 they jointly organized a hands-on tutorial at the international conference Computational Neuroscience (CNS*2019) in Barcelona. This book is a collection of the knowledge we gained from those lectures, hands-on sessions, and preparations, as well as our own research results and experiences. In particular, as a university teacher, I would like to see students write well-structured, clean code with no side effects, rather than just blindly writing code that "just works.

In writing this book, Yamazaki created the framework first, and then the two of us fleshed out the structure. We did not decide on the chapters to be written in advance, so I think the consistency and overall sense of unity is good. However, the first half of Parts I to III were written by Yamazaki, while the rest of the book was written by Igarashi.

0.3 Intended audiences

The intended audience is first of all the general public who are interested in brain computation. The book is basically intended for undergraduates and above, but an introduction to numerical simulations for high school students and various other materials are included in the appendix so that even the most advanced high school students can read it. The authors believe that the book is complete as a reading material, but you will understand it better if you actually run the code and check the changes in behavior by changing the parameters. It is recommended that you skim through Part I and read Part II. I think it is fun to actually reproduce various brain phenomena.

It is also intended for university instructors who teach computational neuroscience and neuroinformatics in their classes. The class will become more dynamic by actually running the system and trying out different parameters, rather than just passing on the knowledge in a classroom. It can also be applied to a flipped classroom, where the text and code are distributed beforehand for the students to try out, and then on the day of the class, a more in-depth lecture is given based on the text and code.

If you are a computational scientist and are interested in spiking network simulation, welcome to this field. After reading Parts I and IV, you will be able to fully demonstrate your expertise in this field. Computational neuroscientists who want to speed up their numerical simulations to make their research more efficient are also important readers. We recommend that you read Part III. I recommend reading Part III, as it will directly relate to your own research results.

0.4 Support page

We have prepared a support page independently from the one of Morikita Publishing.

`https://numericalbrain.org/snsbook/`

You can find errata and the URL of the GitHub where the code is available. Please feel free to send us your thoughts and comments.

0.5 Acknowledgements

Some of the computers and codes used for the numerical simulations were developed by NEDO Core Technology Development for Next-Generation AI and Robotics, MEXT Post-Kyo Budding Researchers Project #4-1 ”Elucidation of Neural Circuit Mechanisms for Thinking and Their Application to Artificial Intelligence,” ”Big Data Analysis of the Brain, Whole-Brain Simulation and Brain-Style Artificial Intelligence Architecture” (hp160246, hp170243, hp180189, hp190146), ”Human-scale whole-brain simulation based on brain connectivity data analysis and functional structure estimation” (hp200139, hp210169) in ”Fugaku” Acceleration Program for the Creation of New Works, Grant-in-Aid for Scientific Research, New Academic Field ”Brain Information Dynamics area ”Multi-region association and parallel processing that define brain information dynamics” (17H06310), Next Generation Area Provided by ”Large-Scale Computational Science with Heterogeneous Many-Core Processors”.

Some of the materials in this book were provided by former members of the Yamazaki Laboratory. Figures 1.1, 1.3 and 6.1 are by Ryohei Hodari. (M.S., March 2021) , the reinforcement learning program in Chapter 7 is from Hideyuki Yoshimura (M.S., March 2020) , and The walking simulation programs in Chapter 9 were provided by Daisuke Ichimura (Ph.D., March 2020). Thank you very much. I wish you all the best in your future endeavors.

We are grateful to Dr. Kenji Doya of Okinawa Institute of Science and Technology Graduate University for his excellent preface. This preface makes the position of this book even clearer. Also, Dr. Hiroshi Yamakawa, President of the Whole Brain Architecture Initiative, gave us a word of recommendation that clearly expresses the purpose of this book. Thank you very much.

Finally, We would like to thank our families for supporting our activities so far.

November 2021 Auspicious Day
The Authors

Part I

Welcome to Spiking Network Simulation

"What I cannot create, I do not understand" – Richard P. Feynmann *[1]

*[1]Richard Feynmann's blackboard at time of his death (Last access: Oct 30 2021), Calisphere. `https://calisphere.org/item/b3e8d3cb9b8adc01314dba1b1f1fcf84/`

Chapter 1

Introduction to computational neuroscience

1.1 What is computational neuroscience?

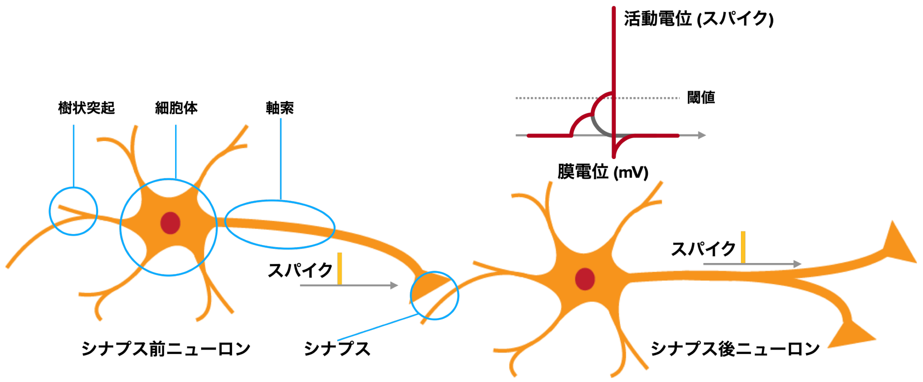


Fig. 1.1 Schematic of a neural network

The brain is composed of neurons, called neurons, and glial cells [64]. Since glial cells are thought to be mainly responsible for environmental maintenance and metabolic support, we will consider neurons in this paper. Neurons behave as electrical elements and have as a parameter a voltage value called the membrane potential. When the membrane potential exceeds a certain value, the neuron fires a short electrical pulse called a spike. When the membrane potential exceeds a certain value, it fires a short electrical pulse called a spike. Neurons are connected to each other in a structure called a synapse, forming a network. When a spike reaches a synapse, it triggers the release of a chemical called a neurotransmitter, and when it reaches the other neuron, an electric current flows (Figure 1.1). Spikes are thought to correspond to 0s or 1s, like bits in a computer*[1], and communicating information using bits is thought to be the principle of information processing in the brain. The human brain has about 86 billion neurons and about

It is said that there are 100 trillion synapses in the brain. It is thought that our brains are responsible for seeing, hearing, speaking, and thinking, and that our brains are generated by networks of neurons. Neuroscience is the field that studies the functions and roles of neurons and networks of neurons.

Neuroscience is a multidisciplinary field and is an amalgamation of various fields such as medicine, biology, chemistry, psychology, physics, and information science. Although the core of neuroscience is the recording of neural activity in humans and other animals, nowadays, neuroscience is not limited to the traditional recording of neural activity from brain slices using electrodes, but also includes

*[1] It is natural to hypothesize that the shape of the spike's waveform itself has a meaning, but we will not go into that in this book.

recording from cultured cells or directly from the whole brain, and non-invasive imaging using laser measurement, fMRI, etc.

In addition, it is possible to genetically modify not only normal animals but also to block specific functions or to give specific functions to specific neurons. Furthermore, it is possible to genetically modify not only normal animals, but also to block certain functions or to give specific functions to certain neurons. The results of brain manipulation are ultimately reflected in the animal's behavior, which also requires behavioral experiments and psychophysical measurements [64].

On the other hand, theoretical neuroscience attempts to clarify the operating principles and mechanisms of the brain through theoretical research*2. In this field, hypotheses about brain mechanisms are formulated and tested, and models of the brain are developed. In this field, we develop models of the brain. A model is a concrete description to test a hypothesis, and the hypothesis is tested by manipulating and evaluating the model under various conditions*3. In theoretical neuroscience, a model is a mathematically abstract description of brain phenomena and the structure of neural circuits, and its behavior is analyzed mathematically or verified by simple numerical simulations. On the other hand, there is a field called computational science, in which numerical simulations are the main research tool. In neuroscience research, with the improvement of computer performance, it has become possible to reproduce brain phenomena and the structure of neural circuits on a computer without abstraction but rather with biological detail, and to verify their behavior by large-scale numerical simulations, in the opposite direction of the conventional models [37 It is now possible to reproduce brain phenomena and the structure of neural circuits on a computer while maintaining the same level of accuracy, and to verify their behavior through large-scale numerical simulation [37]. This is one model that can serve as a tool for testing hypotheses. This approach to neuroscience research is called computational neuroscience*4.

1.2 What is spiking network simulation?

Neural circuit simulation is the numerical simulation of the behavior of neural circuits in the brain on a computer. The functional principles of how the brain works as a network to produce its complex functions are still unclear. On the other hand, a great deal of research has already been done on the single neurons that make up the network, and their behavior can be specifically described by a mathematical equation called a differential equation. Since differential equations can be solved numerically using a computer, it is possible to create a program that calculates neural activity corresponding to that of the human brain, and by using this program, it is possible to attempt to reproduce the human brain on a supercomputer. Various efforts are being made in various countries around the world (details will be discussed in Part IV).

On the other hand, we need to pay attention to the remarkable improvement in the performance of supercomputers. No matter how much we try to recreate the human brain on supercomputers, we will not be able to do so without the memory to store it and the computing power to actually run it. Fortunately, the performance of supercomputers has been improving exponentially over the years and is expected to continue to do so in the future, making the simulation of neural circuits on the scale of a human whole brain a reality. As of June 2021, the number of Japanese supercomputers The Fugaku supercomputer of the University of Tokyo is the world's fastest supercomputer, capable of performing about 44 basic operations per second.

Both theoretical and computational neuroscience start with a mathematical description of the

*2By the way, research in theoretical neuroscience has traditionally been strong in Japan, and a number of excellent textbooks written in Japanese have been published (e.g., [129, 134, 137, 135]). There is no doubt that Japan is a very favorable environment for theoretical research.

*3This is what the authors think about the difference between a hypothesis and a model, but it may be different for some people.

*4The term "computational neuroscience" is more often used in Japanese. This is probably because the research at that time focused on "computational theory," which corresponds to the top of the three levels of Marr.

behavior of neurons and synapses that make up the brain. Furthermore, to properly perform neural circuit simulations, one needs not only knowledge of computational neuroscience, but also knowledge of numerical computation, especially techniques for solving differential equations numerically on a computer.

1.3 Model of neurons

Neurons generally have a characteristic shape and their behavior is complex. The shape of a neuron roughly consists of three parts: (1) **dendrites** that receive input from other neurons, (2) cell body or **soma** that adds up the input, and (3) **axons** that output to other neurons (Figure 1.1). Here, we assume that its essence is to fire spikes by adding inputs from other neurons (We introduce neuron models with spatial structure in Section 3.7).

The basic dynamics of the membrane potential can be described as follows (Gerstner and Kistler, 2002; Dayan and Abbott, 2005)

$$C\frac{dV}{dt} = -\overline{g}_{\mathrm{leak}}\left(V(t) - E_{\mathrm{leak}}\right) + I_{\mathrm{ext}}(t) \tag{1.1}$$

where $V(t)$ is the membrane potential at time tt (mV), $I_{\mathrm{ext}}(t)$ is the current flowing in from the outside (μA/cm^2), and C, $\overline{g}_{\mathrm{leak}}$, and E_{leak} leakare constants, called **capacitance** (μF/cm^2), **conductance** (inverse of resistance) (mS/cm^2), and **reversal potential** (mV), respectively. The reason for the 1/cm^2 unit is that the values are per unit area of the cell membrane. This formulation is equivalent to thinking of a neuron as an electrical circuit as shown in Figure 1.2A. In this formulation, the value of the membrane potential $V(t)$ is at E_{leak} in the absence of external current, and converges to $V(t) = E_{\mathrm{leak}} + I_{\mathrm{ext}}/\overline{g}_{\mathrm{leak}}$ with a **time constant** $\tau_m = C/\overline{g}_{\mathrm{leak}}$ when a constant current I_{ext} is applied. When the external current is stopped, $V(t)$ returns to E_{leak} again. An increase in the value of the membrane potential is called **depolarization**, and a decrease in the value is called **hyperpolarization**.

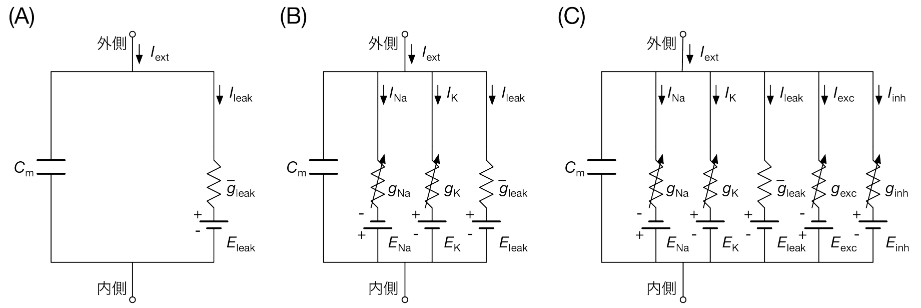


Fig. 1.2 Electrical circuits equivalent to a neuron. (A) RC circuit corresponding to the cell membrane. (B) Spike-firing circuit with additional Na$^+$ channels and K$^+$ channels. The conductances of Na$^+$ channels and K$^+$ channels are variables, so the diagonal arrowhead symbols wer drawn above the resistances. (C) Circuit with additional synaptic currents. I_{exc}, g_{exc}, and E_{exc} are excitatory synaptic currents, conductances, and reversal potentials, repectively, and I_{inh}, g_{inh}, and E_{inh} are the inhibitory ones.

This alone merely causes the membrane potential to rise and fall, but does not fire spikes. Spikes are generated mainly by currents generated by Na$^+$ ions and K$^+$ ions passing through holes called **ion channels** on the surface of the membrane. Those currents, $I_{\mathrm{Na}}(t)$ and $I_{\mathrm{K}}(t)$, are given by

$$\begin{aligned} I_{\mathrm{Na}}(t) &= -g_{\mathrm{Na}}(V,t)\left(V(t) - E_{\mathrm{Na}}\right) \\ I_{\mathrm{K}}(t) &= -g_{\mathrm{K}}(V,t)\left(V(t) - E_{\mathrm{K}}\right) \end{aligned} \tag{1.2}$$

where $g_{\mathrm{Na}}(V,t)$ and $g_{\mathrm{K}}(V,t)$ are the Na^+ channel and K^+ channel conductance, respectively, and a function of time and membrane potential (**voltage dependence**). Moreover, E_{Na} and E_{K} are **reversal potential** of each ion channel. The difference between the membrane potential and the reversal potential is the external force, and the product of this and the conductance is the current flowing through the channel. Since conductance is the reciprocal of resistance, these two equations are equivalent to Ohm's law ($V = RI$). The change in the conductance value causes a rapid rise and fall in the membrane potential value over time, creating a spike. An equivalent circuit incorporating Equation (1.2) into Equation (1.1) is shown in Figure 1.2B. The conductance and reversal potential of the leakage current (the first term on the right side of Equation (1.1)) are mainly determined by the Cl^-. When all of the Cl^-, Na^+, and K^+ currents in Eq. (1.1) are taken into account, the membrane potential takes a value around -65 mV when there is no external current and the neuron is at rest. This value is called the **resting potential**.

The conductance equation is different for different species of animals and different types of neurons. Hodgkin and Huxley were the first in the world to describe the conductance equation and reveal the mechanism of spike firing, but they did so using the neurons of the squid (Hodgkin and Huxley, 1952). In Section 3.1, we introduce the actual equation and parameters, and perform numerical simulations.

Now that we've covered neurons, let's move on to synapses.

1.4 Models of synapse

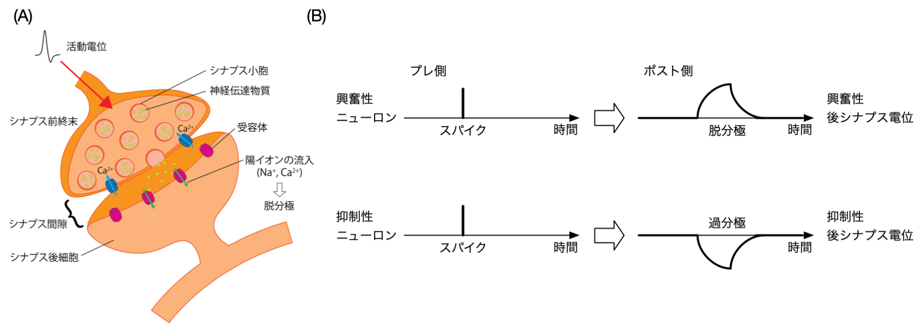


Fig. 1.3 Synapse structure and function. (A) Schematic representation of a synapse. (B) The generation of postsynaptic potentials at a synapse.

Synapses, as mentioned earlier, are the connection points of neurons and the transmission points of signals. It is the point where the axon terminal of the sender (pre-side) neuron and the dendrite of the receiver (post-side) neuron connect (Figure 1.3A). However, they are not in physical contact, but rather have a small space between them (the **synaptic cleft**) When a pre-synaptic neuron fires a spike, the signal propagates to the axon terminal and neurotransmitters are released into the inter-synaptic gap. There are two main types of neurons: **excitatory neurons** and **inhibitory neurons**, which release different neurotransmitters. When the pre-side neuron is an excitatory neuron (excitatory synapse), it releases **glutamate**, and when it is an inhibitory neuron (inhibitory synapse), it releases **GABA** (γ aminobutyric acid). On the other hand, posterior neurons have **receptors** for neurotransmitters on the surface of their dendrites, and when they receive a neurotransmitter, they activate ion channels to generate an electric current (**synaptic current**). Excitatory synapses produce depolarizing currents (**excitatory postsynaptic currents**, **EPSCs**), and inhibitory synapses produce hyperpolarizing currents (**inhibitory postsynaptic currents**, **IPSCs**). This causes a change in the membrane potential of the posterior neuron. These are called **excitatory postsynaptic potentials** (**EPSPs**) and **inhibitory postsynaptic potentials** (**IPSPs**), respectively (Fig. 1.3B).

The synaptic current $I_{\mathrm{syn}}(t)$, like the ion current, is calculated using the synaptic conductance $g_{\mathrm{syn}}(t)$ as follows

$$I_{\mathrm{syn}}(t) = -g_{\mathrm{syn}}(t)\,(V(t) - E_{\mathrm{syn}}) \tag{1.3}$$

Here, E_{syn} is the reversal potential of the synapse, and values such as 0 mV for excitatory synapses and -65 to -80 mV for inhibitory synapses are mostly used (when the resting potential is -65 mV). This current is applied to the post-side neuron. An equivalent electrical circuit is shown in Figure 1.2C. The value of $g_{\mathrm{syn}}(t)$ changes when a spike is received from the pre-side neuron, and decays to zero if there is none. The time constant of the change in value is determined by the type of synapse. In the case of excitatory synapses, there are two major types: **AMPA** (α-amino-3-hydroxy-5-mesoxazole-4-propionic acid) and **NMDA** (N-methyl-D-aspartate). AMPA and NMDA receptors have short (a few ms) and long (tens to hundreds ms) time constants, respectively. Although omitted from equation (1.3), the NMDA type has a mechanism called voltage-dependent magnesium block, which is thought to play an important role in brain learning. In the case of inhibitory synapses, they are divided into GABA$_A$ and GABA$_B$, the former having a short time constant and the latter a long time constant.

As explained above, receptors that are directly activated by the binding of transmitters are called **ionotropic** type. In addition, there are other types of receptors called **metabotropic** receptors that are activated indirectly through several different receptor pathways. In addition, there is a different type of coupling called **gap junctions**, in which neurons are electrically coupled directly to each other rather than through neurotransmitters. The latter is an important topic, especially in relation to synchronizing the activity of a population of neurons or simulating neurons with spatial geometry, but it is beyond the scope of this book and will not be discussed here. For details, please refer to Dayan and Abbott (2005); Gerstner and Kistler (2002); Gerstner et al. (2014).

This is a rough description of synapses. Later in Section 3, we will discuss how to make a synaptic network that can be used in a variety of ways.

1.5 Column: To use, or not to use, that is a simulator

The goal of this book is to enable you to write your own simulation code from scratch, but in general, you will use dedicated simulator software. For example, NEST (Gewaltig and Diesmann, 2007), Brian (Goodman and Brette, 2008), and GeNN (Yavuz et al., 2016) are typical simulators for single-compartment models, and NEURON (Carnevale and Hines, 2006) and Arbor (Akar et al., 2019) are typical simulators for multi-compartment models. These simulators have easy-to-use interfaces, are equipped with optimized and fast numerical algorithms, and have been tested, so they can be used quickly and provide immediate results. Especially now that it has a Python interface, it can be used from within Jupyter Notebook, which is very convenient. Recently, BindsNET (Hazan et al., 2018), a simulator based on TensorF low, a framework for deep learning, has also appeared, and it is a field that we cannot take our eyes off.

If you do, you will not have to write the code from scratch anymore. On the other hand, what is the situation where you have to write the code from scratch?

The first situation that is easy to understand is when the simulator cannot be used. For example, when a new supercomputer is built, it is necessary to port the simulator on it, but it is not always easy because the compatibility is often low due to special mechanisms for speedup. Recently, the authors ported NEST 2 with PyNEST and MPI support on a computation node of Fugaku, and managed to install the necessary tools and libraries first, and then dealt with difficult error messages and behaviors during compilation. However, this was a fortunate example, and there were many cases where the compilation did not pass in the first place. On the other hand, if you write your own code, it is often relatively easy to port because you only need the minimum amount of code. The fact that the model of the neuron or synapse you want to use, or the synaptic plasticity (see Section 4.2) has not yet been implemented in the simulator, is also an incentive to develop your own code.

Next, even if a simulator is installed, it does not mean that a high-speed numerical simulation utilizing the supercomputer's functions can be realized immediately. For example, "Fugaku" has a SIMD function called Scalable Vector Extension (SVE) (see Section 16.1), but the simulator must be rewritten to support it, which naturally requires considerable effort. If it is your own code, you know the structure of the code well, so the effort to rewrite it is greatly reduced.

Furthermore, even though the numerical methods used in simulators are highly optimized, they can only be designed to provide a reasonable performance for any network, and thus cannot be optimized to take the characteristics of the network into account. Especially in parallel computing, the parallelization method makes a real difference. For example, a completely random network and a network structured on tiles require different partitioning and parallelization methods (Igarashi et al., 2019). If you write your own code, you can choose the best parallelization and numerical method according to the network you want to simulate.

The most important thing to remember is that you will be able to understand the problem better by creating it yourself, and if you do not do so, you will fall into various mistakes. For example, if you choose the Euler method without knowing about numerical accuracy, and if you take a large value of Δt because the calculation is fast, and if you run a simulation with a large error, the result will not be reliable [*5] . It is essential to be familiar enough with modeling and methods to be able to write from scratch in order to proceed with research with confidence. In general, I can't write methods when I write a paper.

Finally, I would like to point out that in order to participate in the development of such a simulator, it is essential to have the ability to write the simulation code from scratch.

From the above, the author's conclusion is that it is not always necessary to write from scratch,

[*5] It is more likely that the calculation will be blown up in the first place.

but the progress of research changes depending on whether one can actually write when necessary. Especially in the case of researchers, if the way of research progresses changes, the rest of their lives will also change.

Chapter 2

Numerical solution of ordinary differential equations

This chapter provides an introduction to numerical simulations in general. Readers who are already familiar with numerical methods for solving ordinary differential equations can skip this chapter. If you are not, you may be disappointed to know that we are going to leave the topic of the brain once, but please look forward to the next chapter where we will finally start the neural circuit simulation.

2.1 Initial value problems for ordinary differential equations

Let $x(t)$ be a variable with respect to time t, and let its time variation be given by $f(x, t)$.

$$\begin{aligned} \frac{dx}{dt} &= f(x,t) \\ x(0) &= x_0 \end{aligned} \tag{2.1}$$

where, $x(0) = x_0$ is the initial value of x at time 0, and is called the **initial condition**. Equation (2.1) has only one free variable, t. Such a differential equation is called an **ordinary differential equation (ODE)**. The problem of solving a differential equation under a given initial conditiong is called an initial value problem.In the equation (1.1) for the membrane potential of a neuron, the variables $x(t) = V(t)$, $f(x,t) = \left(-\bar{g}_{\text{leak}}\left(V(t) - E_{\text{leak}}\right) + I_{\text{ext}}(t)\right)/C$.

2.2 Euler method

The simplest way to solve equation (2.1) numerically is to use the (explicit) **Euler method**. Mathematically, the definition of differentiation is

$$\frac{dx}{dt} = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} \tag{2.2}$$

However, the computer cannot handle infinitely small values, so instead, we consider a sufficiently small value Δt, and compute

$$\frac{dx}{dt} \approx \frac{x(t + \Delta t) - x(t)}{\Delta t} \tag{2.3}$$

It is approximated as Transforming this equation, we obtain

$$x(t + \Delta t) = x(t) + \Delta t f(x,t) \tag{2.4}$$

is obtained. Since the initial value $x(0) = x_0$ is given, we can calculate the values of $x(t)$ from here in turn in Δt increments as $x(0)$, $x(\Delta t)$, $x(2\Delta t)$, $\cdots$.

Let time be discretized by Δt as $t_0 = 0$, $t_1 = \Delta t$, $t_2 = 2\Delta t$, $\cdots$ and let the values of $x(t)$ at each time by $x_0 = x(0)$, $x_1 = x(t_1)$, $x_2 = x(t_2)$, $\cdots$. In this case, the Euler method can be written as

$$\begin{aligned} k_1 &= \Delta t f(x_n, t_n) \\ x_{n+1} &= x_n + k_1 \end{aligned} \tag{2.5}$$

2.2.1 Error analysis of Euler method

The Euler method is a simple but **inaccurate** method; consider the Taylor expansion of $x(t + \Delta t)$ around t

$$\begin{aligned} x(t+\Delta t) &= x(t) + \frac{1}{1!}f(x,t)\Delta t + \frac{1}{2!}f'(x,t)\Delta t^2 + \cdots \\ &= x(t) + \frac{1}{1!}f(x,t)\Delta t + O\left(\Delta t^2\right) \end{aligned} \tag{2.6}$$

The result is Here, $O(x)$ is in order notation, where $O(x) \leq cx + d$ for any x with some constants c, d [*1]. Comparing the right-hand sides of equations (2.5) and (2.6), we see that they agree up to the first order term of Δt, but ignores the second-order and subsequent terms. The **error** of the Euler method is

$$\begin{aligned} \frac{x(t+\Delta t) - x(t)}{\Delta t} - f(x,t) &= \frac{1}{\Delta t}\left(x(t) + \frac{1}{1!}x'(t)\Delta t + \frac{1}{2}x''(t)\Delta t^2 + \cdots - x(t)\right) - f(x,t) \\ &= x'(t) + \frac{1}{2!}x''(t)\Delta t - f(x,t) \\ &= \frac{1}{2!}x''(t)\Delta t = O(\Delta t) \end{aligned} \tag{2.7}$$

and is proportioal to Δt. This is said to be a first-order precision calculation method. This means that when the value of Δt is increased to 1/100, the error also increases by a factor of 1/100.

2.3 Heun's method

Thus, the computational accuracy of the numerical solution of ordinary differential equations depends on how many order terms of the Taylor expansion can be matched? The **Heun's method** is a second-order accurate solution method that matches up to the second-order term of Δt. It is calculated as follows

$$\begin{aligned} k_1 &= \Delta t f(x_n, t_n) \\ k_2 &= \Delta t f(x_n + k_1, t_n + \Delta t) \\ x_{n+1} &= x_n + \frac{k_1 + k_2}{2} \end{aligned} \tag{2.8}$$

The Euler method used the slope to calculate the value one step ahead. Heun's method calculates the value one more step ahead and takes the average. Heun's method is sometimes referred to as the second-order Runge-Kutta method.

2.3.1 Error analysis of Heun's method

Let's also analyze the accuracy of the Heun's method. Consider the Taylor expansion again.

$$x(t+\Delta t) = x(t) + \frac{1}{1!}x'(t)\Delta t + \frac{1}{2!}x''(t)\Delta t^2 + O(\Delta t^3) \tag{2.9}$$

[*1] This means that in equation (2.6), the residuals are only about a constant multiple of Δt^2.

This time we will calculate $x''(t)$ properly, from $\dfrac{dx}{dt} = f(x,t)$

$$x''(t) = \frac{d}{dt}f(x,t) = \frac{\partial}{\partial t}f(x,t) + \frac{\partial}{\partial x}f(x,t)x'(t) \tag{2.10}$$

(Differentiation of a bivariate function)

First, we prepare a discretized expression for the Taylor expansion.

$$x_{i+1} = x_i + \frac{1}{1!}x'(t_i)\Delta t + \frac{1}{2!}x''(t_i)\Delta t^2 + O(\Delta t^3) \tag{2.11}$$

Also, from the Taylor expansion of the two variables,

$$f(x+\alpha\Delta t, t+\beta\Delta t) = f(x,t) + \alpha\Delta t\frac{\partial}{\partial x}f(x,t) + \beta\Delta t\frac{\partial}{\partial t}f(x,t) + O(\Delta t^2) \tag{2.12}$$

Substituting Eq. (2.10) into Eq. (2.12), with $\alpha = f(x_i, t_i)$ and $\beta = 1$, we obtain

$$\begin{aligned} f(x_i + f(x_i,t_i)\Delta t, t_i + \Delta t) &= f(x_i,t_i) + \Delta t f(x_i,t_i)\frac{\partial}{\partial x}f(x_i,t_i) + \Delta t\frac{\partial}{\partial t}f(x_i,x_t) + O(\Delta t^2) \\ &= f(x_i,t_i) + \Delta t x''(t_i) + O(\Delta t^2) \end{aligned} \tag{2.13}$$

In the last transformation we used $f(x_i,t_i) = x'(t_i)$ from $\dfrac{dx}{dt} = f(x,t)$. Substituting equation (2.13) into equation (2.11), we get

$$\begin{aligned} x_{i+1} &= x_i + \frac{1}{1!}x'(t_i)\Delta t + \frac{1}{2!}x''(t_i)\Delta t^2 + O(\Delta t^3) \\ &= x_i + \frac{1}{1!}x'(t_i)\Delta t + \frac{1}{2!}\left(\frac{f(x_i + \Delta t f(x_i,t_i), t_i + \Delta t) - f(x_i,t_i) - O(\Delta t^2)}{\Delta t}\right)\Delta t^2 + O(\Delta t^3) \\ &= x_i + \frac{\Delta t}{2}\left(f(x_i,t_i) + f(x_i + \Delta t f(x_i,t_i), t_i + \Delta t)\right) + O(\Delta t^3) \end{aligned} \tag{2.14}$$

In the last variant, $f(x_i,t_i) = x'(t)$ from $\dfrac{dx}{dt} = f(x,t)$. Finally, $k_1 = f(x_i,t_i)$, $k_2 = f(x_i + \Delta t f(x_i,t_i), t_i + \Delta t)$, Equation (2.8) can be obtained by setting. Since the residuals are $O(\Delta t^3)$, the accuracy of the calculation is second order. This means that if we set Δt to 1/100, we get in the Heun's method, the error is $1/100^2 = 1/10000$. Euler's method means that even if we set Δt to 1/100, the error is still 1/100. This indicates that the Heun's metho has a better accuracy.

2.4 The Runge-Kutta method

Thus, as more variables with intermediate values are added, the accuracy can be further increased. The (fourth-order) **Runge-Kutta method** is a numerical solution method with fourth-order accuracy and is the most widely used method for numerical computation of ordinary differential equations. It can be computed as follows

$$\begin{aligned} k_1 &= \Delta t f\left(x_n, t_n\right) \\ k_2 &= \Delta t f\left(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2}\right) \\ k_3 &= \Delta t f\left(x_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2}\right) \\ k_4 &= \Delta t f\left(x_n + k_3, t_n + \Delta t\right) \\ x_{n+1} &= x_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned} \tag{2.15}$$

We will leave it to other textbooks (e.g., 皆本 (2005)) to explain why this is fourth-order accurate, due to the length of the document.

$$x(t+\Delta t) = x(t) + \frac{1}{1!}x'(t)\Delta t + \frac{1}{2!}x''(t)\Delta t^2 + \frac{1}{3!}x^{(3)}(t)\Delta t^3 + \frac{1}{4!}x^{(4)}(t)\Delta t^4 + O(\Delta t^5) \tag{2.16}$$

and Δt to fourth order, and then expand the Taylor expansion of $k_1, \cdots, k_4$ to fourth order as in the Heun's method.

In general, the Runge-Kutta method can be used for more stages and the accuracy can be improved, but the fourth-order method is the most popular because of the balance between accuracy and implementation.

2.5 Evaluation of errors by various methods

Table 2.1 The value of e obtained by solving equation (2.17). The parts with matching digits are in bold.

Δt	Euler method	Heun's method	Runge-Kutta method
1.000000000000000	**2**.000000000000000	**2**.500000000000000	**2.7**08333333333333
0.500000000000000	**2**.250000000000000	**2**.640625000000000	**2.71**7346191406250
0.250000000000000	**2**.441406250000000	**2**.694855690002441	**2.7182**09939201323
0.125000000000000	**2**.565784513950348	**2.71**1841238551985	**2.7182**76844416735
0.062500000000000	**2**.637928497366600	**2.71**6593522474767	**2.718281**500340586
0.031250000000000	**2**.676990129378183	**2.71**7849673980259	**2.7182818**07411193
0.015625000000000	**2**.697344952565100	**2.718**172511563830	**2.71828182**7126323
0.007812500000000	**2.7**07739019688019	**2.7182**54338321275	**2.718281828**375204
0.003906250000000	**2.71**2991624253433	**2.7182**74935740745	**2.7182818284**53785
0.001953125000000	**2.71**5632000168990	**2.71828**0102752167	**2.7182818284**58716
0.000976562500000	**2.71**6955729466436	**2.718281**396716139	**2.718281828459**026
0.000488281250000	**2.71**7618482336880	**2.718281**720483778	**2.718281828459**050

Let's investigate the errors of Euler, Heun's, and Runge-Kutta methods using a real-world example. Consider the following initial value problem.

$$\begin{aligned} \frac{dx}{dt} &= x \\ x(0) &= 1.0 \end{aligned} \tag{2.17}$$

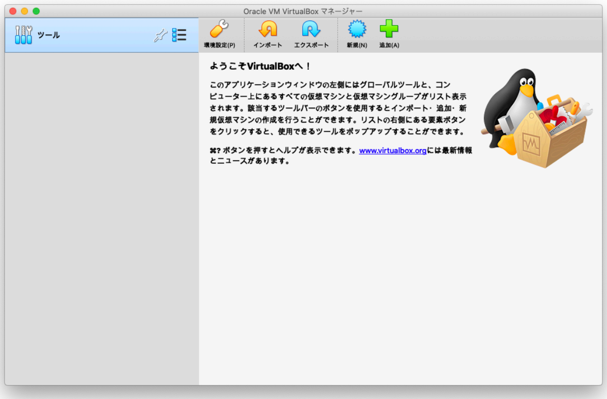
Since the exact solution is $x(t) = \exp(t)$, it should take the value $x(1) = e = 2.718281828459045\cdots$. Table 2.1 summarizes the values (double precision) of $x(1)$ for different time increments Δt, decreasing by 1/2 from 1.0. In the case of Euler method, only 3 digits are matched even when $\Delta t = 2^{-11}$, but in the case of Runge-Kutta method, up to 14 digits are matched. However, Δt is made smaller than that, the **rounding error**[*2] becomes larger, and the error becomes larger in the opposite direction. Thus, it is important to note that the smaller Δt is not always the better.

[*2]Since the representation of a numerical value by a computer is finite, it refers to the error that occurs in keeping a numerical value within a certain number of digits.

2.6 Column: How to prepare a linux environment

In order to try out the simulations described in the next chapter, we need at least a C compiler and some kind of graphing application, and more generally, we need a development environment. If you are using Windows, you can install Visual Studio or Windows Subsystem for Linux (WSL), and if you are using Mac, you can install XCode or HomeBrew to set up a development environment, but since almost all supercomputers today, including "Fugaku", are Linux machines, it is convenient to be familiar with Linux. It's hard to prepare a brand-new PC, so let's prepare a Linux virtual environment. Ubuntu is becoming the de facto standard, so I'm going to install Ubuntu 20.04 (LTS).

There are many virtualization applications, but one of the free ones is Oracle VirtualBox [*3]. I installed it on my MacBook Pro (10.15.5 Catalina). You can download the dmg file for OS X hosts from the "Download VirtualBox " [*4] website and follow the instructions to install it as usual. For Windows, you can download the exe file for Windows hosts.



ようこそVirtualBoxへ！

このアプリケーションウィンドウの左側にはグローバルツールと、コンピューター上にあるすべての仮想マシンと仮想マシングループがリスト表示されます。該当するツールバーのボタンを使用するとインポート・追加・新規仮想マシンの作成を行うことができます。リストの右側にある要素ボタンをクリックすると、使用できるツールをポップアップすることができます。

⌘? ボタンを押すとヘルプが表示できます。www.virtualbox.orgには最新情報とニュースがあります。

Fig. 2.1 Oracle VM VirtualBox Manager screen.

After the installation is complete, launch it and the "Oracle VM VirtualBox Manager" window (Figure 2.1) will open. Select "New". Enter "Ubuntu 20.04 LTS" in the "Name" field, "Linux" in the "Type" field, "Ubuntu (64-bit)" in the "Version" field, and click "Continue". Next, you will be asked for the "Memory Size", set it to any value you like. In this case, I chose "1024 MB". Next, you will be asked about the "Hard Disk", select "Create Virtual Hard Disk" and click "Create". Then, you will be asked about the "Hard Disk File Type", select "VDI (VirtualBox Disk Image)". Next, you will be asked about "Storage on physical hard disk", select "Variable size". Finally, you will be asked about "File location and size", leave it at the default and click "Create". Then, a virtual machine with no OS installed will be configured. Before clicking "Boot" and turning on the power, get the image of the OS.

ISO images of Ubuntu 20.04 (LTS) can be obtained from the Ubuntu Japanese Team [*5] website [*6]. Note that the file size is large. After obtaining the ISO image, go back to the VirtualBox window and click "Start". When the tab "No OS, specify a file" opens, specify the ISO image you just obtained. Click "Start" and you will see the following The virtual machine will start and boot with the ISO image. Select "Install Ubuntu" and follow the instructions to proceed with the installation. When the installation process is finished, you will be asked to reboot, and Ubuntu will boot successfully.

[*3] `https://www.virtualbox.org/`
[*4] `https://www.virtualbox.org/wiki/Downloads`
[*5] `https://www.ubuntulinux.jp/`
[*6] `https://www.ubuntulinux.jp/News/ubuntu2004-ja-remix`

Skip "Connect to online account" for now, "Next" for LivePatch [*7], and "Help improve Ubuntu". Select "Privacy" as you like. When it's done, "Software Update" will be waiting for you, so "Install Now". In the meantime Go to "Settings" and "Display" to adjust the screen resolution. After the software update is finished, the system will reboot once.

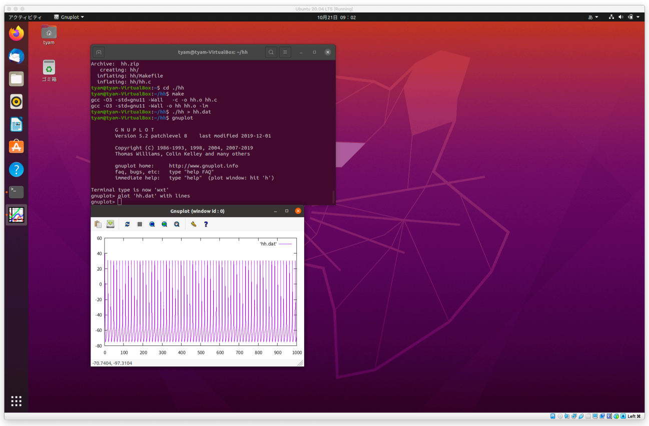
After rebooting, select "Terminal" from the launcher in the lower left corner and start it up. At this point, there is no development environment, so install gcc, make, gnuplot, and an editor. We will use emacs as the editor.

```
user@user-VirtualBox:~$ sudo apt install gcc make gnuplot-x11 emacs-gtk
```

We have prepared the code for the Hodgkin-Huxley model (see section 3.1) for testing, so let's use it to test the behavior.

```
user@user-VirtualBox:~$ wget https://numericalbrain.org/wp-content/uploads/snsbook/hh.zip
user@user-VirtualBox:~$ unzip hh.zip
user@user-VirtualBox:~$ cd ./hh
user@user-VirtualBox:~$ make
user@user-VirtualBox:~$ ./hh > hh.dat
user@user-VirtualBox:~$ gnuplot
gnuplot> plot 'hh.dat' with lines
```

Did you see the spike waveform well displayed as shown in Figure 2.2?



```
Archive:  hh.zip
   creating: hh/
  inflating: hh/Makefile
  inflating: hh/hh.c
tyam@tyam-VirtualBox:~$ cd ./hh
tyam@tyam-VirtualBox:~/hh$ make
gcc -O3 -std=gnu11 -Wall   -c -o hh.o hh.c
gcc -O3 -std=gnu11 -Wall -o hh hh.o -lm
tyam@tyam-VirtualBox:~/hh$ ./hh > hh.dat
tyam@tyam-VirtualBox:~/hh$ gnuplot

        G N U P L O T
        Version 5.2 patchlevel 8    last modified 2019-12-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2019
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'wxt'
gnuplot> plot 'hh.dat' with lines
gnuplot>
```

Fig. 2.2 Screenshot of the terminal after booting and testing the operation as per the procedure.

[*7] I need to create an account for Ubuntu One.

Chapter 3

Introduction to spiking network simulation

In this section, you will learn how to write a simulation program after understanding the differential equation description of neurons and synapses and the numerical solution of ordinary differential equations. The sections and paragraphs marked with an asterisk (*) in this book are for advanced readers, so beginners may skip them. The code used in this and subsequent chapters can be obtained from GitHub, which is referenced on the support page.

3.1 Simulation of the Hodgkin-Huxley model

3.1.1 Hodgkin-Huxley model

The world's first mathematical model of a single neuron was formulated by Alan L. Hodgkin and Andrew F. Huxley [*1] (Hodgkin and Huxley, 1952). They performed a detailed analysis using electrophysiological recordings from giant axons of the giant squid, and showed that the current of Na^+ ions and K^+ ions, especially a mechanism called **voltage-dependent** conductance, plays an important role. He also derived a differential equation describing the time variation of the conductance value, and drew the waveform using a hand-cranked calculator (!) [*2] (Schwiening, 2012). This so-called Hodgkin-Huxley (HH) model is the basis of all neuron models.

The HH model is expressed by the following equation (Hodgkin and Huxley, 1952; Gerstner and Kistler, 2002).

$$C\frac{dV}{dt} = -\overline{g}_{\mathrm{leak}}\left(V(t) - E_{\mathrm{leak}}\right) - g_{\mathrm{Na}}(V,t)\left(V(t) - E_{\mathrm{Na}}\right) - g_{\mathrm{K}}(V,t)\left(V(t) - E_{\mathrm{K}}\right) + I_{\mathrm{ext}}(t) \tag{3.1}$$

where C is the membrane capacitance (μF/cm^2), t is (ms), $V(t)$ is the membrane potential (mV), $\overline{g}_{\mathrm{leak}}$ is a conctant leakage conductance (mS/cm^2), E_{leak} is the reversal potential of mainly Cl^- ions (mV), $g_{\mathrm{Na}}(V,t)$ is the voltage-dependent conductance of Na^+ channels (mS/cm^2), E_{Na} is the reversal potential of Na^+ ions (mV), $g_{\mathrm{K}}(V,t)$ is the voltage-dependent conductance of K^+ channels (mS/cm^2), E_{K} is the reversal potential of K^+ ions (mV), $I_{\mathrm{ext}}(t)$ is the current injected from outside the cell (μA/cm^2). Since the total amounts of capacitance and channels are proportional to the membrane surface, these values are normalized per surface. A circuit equivalent to the HH model is already shown in Figure 1.2B.

$g_{\mathrm{Na}}(V,t)$ and $g_{\mathrm{K}}(V,t)$ are calculated by the following equations, respectively

$$\begin{aligned} g_{\mathrm{Na}}(V,t) &= \overline{g}_{\mathrm{Na}} m^3(V,t)h(V,t) \\ g_{\mathrm{K}}(V,t) &= \overline{g}_{\mathrm{K}} n^4(V,t) \end{aligned} \tag{3.2}$$

[*1] This paper has been published in PDF. `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/pdf/jphysiol01442-0106.pdf`
[*2] This paper is also available in PDF. `https://physoc.onlinelibrary.wiley.com/doi/10.1113/jphysiol.2012.230458`

where $\overline{g}_{\mathrm{Na}}$ and $\overline{g}_{\mathrm{K}}$ are constants for maximal conductances (mS/cm^2), respectively, and $m(V,t), h(V,t)$, and $n(V,t)$ are **gate variables** depending on the membrane potential V, represent the aperture of the ion channels and are updated by the following equations

$$\begin{aligned}
\frac{dm}{dt} &= \alpha_m(V)\,(1-m(V,t)) - \beta_m(V)m(V,t) \\
\frac{dh}{dt} &= \alpha_h(V)\,(1-h(V,t)) - \beta_h(V)h(V,t) \\
\frac{dn}{dt} &= \alpha_n(V)\,(1-n(V,t)) - \beta_n(V)n(V,t)
\end{aligned} \tag{3.3}$$

$\alpha_x(V)$ and $\beta_x(V)$ are defined by the following equations, respectively.

$$\begin{aligned}
\alpha_m(V) &= \frac{2.5-0.1V}{\exp(2.5-0.1V)-1} \\
\beta_m(V) &= 4\exp\left(-\frac{V}{18}\right) \\
\alpha_h(V) &= 0.07\exp\left(-\frac{V}{20}\right) \\
\beta_h(V) &= \frac{1}{\exp(3-0.1V)+1} \\
\alpha_n(V) &= \frac{0.1-0.01V}{\exp(1-0.1V)-1} \\
\beta_n(V) &= 0.125\exp\left(-\frac{V}{80}\right)
\end{aligned} \tag{3.4}$$

All equations are given above. The constants are $\overline{g}_{\mathrm{leak}} = 0.3$ mS/cm^2, $E_{\mathrm{leak}} = 10.6$ mV, $\overline{g}_{\mathrm{Na}} = 120$ mS/cm^2, $E_{\mathrm{Na}} = 115$ mV, $\overline{g}_{\mathrm{K}} = 36$ mS/cm^2, $E_{\mathrm{K}} = -12$ mV when normalized to $C = 1$ μF/cm^2.

The following is a supplement to equation (3.3) for gate variables. By transforming the equations, we obtain

$$\tau_x(V)\frac{dx}{dt} = -(x - x_0(V)) \tag{3.5}$$

where $\tau_x(V) = 1.0/(\alpha_x(V)+\beta_x(V))$, $x_0(V) = \alpha_x(V)/(\alpha_x(V)+\beta_x(V))$. In other words, in steady state, the value of x is $x_0(V)$, which varies with time constant $\tau_x(V)$.

In summary, the HH model is a differential equation consisting of four variables (V, n, m, h), which can be solved numerically to reproduce the behavior of neurons, especially the behavior of spike firing.

3.1.2 Simulation of HH model

Let's try to simulate one neuron of the HH model. Equations (3.1)–(3.5) can be written as List 3.1. We show the entire code, because this is the starting point for all other neuron models. The code can be found at `code/part1/hh/`.

Listing 3.1 **`hh.c`**

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>

#define E_REST ( -65.0 ) // mV
#define C      (   1.0 ) // micro F / cm^2
```

```c
#define G_LEAK (   0.3 ) // mS / cm^2
#define E_LEAK (  10.6 + ( E_REST ) ) // mV
#define G_NA   ( 120.0 ) // mS / cm^2
#define E_NA   ( 115.0 + ( E_REST ) ) // mV
#define G_K    (  36.0 ) // mS / cm^2
#define E_K    ( -12.0 + ( E_REST ) ) // mV

#define DT ( 0.01 ) // 10 micro s
#define T  ( 1000 ) // 1000 ms; unused
#define NT ( 100000 ) // T / DT

static inline double alpha_m ( const double v ) { return ( 2.5 - 0.1 * ( v - E_REST ) ) / ( exp ( 2.5 - 0.1 * ( v - E_REST ) ) - 1.0 ); }
static inline double beta_m  ( const double v ) { return 4.0 * exp ( - ( v - E_REST ) / 18.0 ); }
static inline double alpha_h ( const double v ) { return 0.07 * exp ( - ( v - E_REST ) / 20.0 ); }
static inline double beta_h  ( const double v ) { return 1.0 / ( exp ( 3.0 - 0.1 * ( v - E_REST ) ) + 1.0 ); }
static inline double alpha_n ( const double v ) { return ( 0.1 - 0.01 * ( v - E_REST ) ) / ( exp ( 1 - 0.1 * ( v - E_REST ) ) - 1.0 ); }
static inline double beta_n  ( const double v ) { return 0.125 * exp ( - ( v - E_REST ) / 80.0 ); }

static inline double m0 ( const double v ) { return alpha_m ( v ) / ( alpha_m ( v ) + beta_m ( v ) ); }
static inline double h0 ( const double v ) { return alpha_h ( v ) / ( alpha_h ( v ) + beta_h ( v ) ); }
static inline double n0 ( const double v ) { return alpha_n ( v ) / ( alpha_n ( v ) + beta_n ( v ) ); }
static inline double tau_m ( const double v ) { return 1. / ( alpha_m ( v ) + beta_m ( v ) ); }
static inline double tau_h ( const double v ) { return 1. / ( alpha_h ( v ) + beta_h ( v ) ); }
static inline double tau_n ( const double v ) { return 1. / ( alpha_n ( v ) + beta_n ( v ) ); }

static inline double dmdt ( const double v, const double m ) { return ( 1.0 / tau_m ( v ) ) * ( - m + m0 ( v ) ); }
static inline double dhdt ( const double v, const double h ) { return ( 1.0 / tau_h ( v ) ) * ( - h + h0 ( v ) ); }
static inline double dndt ( const double v, const double n ) { return ( 1.0 / tau_n ( v ) ) * ( - n + n0 ( v ) ); }
static inline double dvdt ( const double v, const double m, const double h, const double n, const double i_ext )
{
  return ( - G_LEAK * ( v - E_LEAK ) - G_NA * m * m * m * h * ( v - E_NA ) - G_K * n * n * n * n * ( v - E_K ) + i_ext ) / C;
}

int main ( void )
{
  double v = E_REST;
  double m = m0 ( v );
  double h = h0 ( v );
  double n = n0 ( v );

  double i_ext = 9.0; // micro A / cm^2

  for ( int32_t nt = 0; nt < NT; nt++ ) {
    double t = DT * nt;
    printf ( "%f %f %f %f %f\n", t, v, m, h, n );

    double dmdt1 = dmdt ( v, m );
    double dhdt1 = dhdt ( v, h );
    double dndt1 = dndt ( v, n );
    double dvdt1 = dvdt ( v, m, h, n, i_ext );

59       double dmdt2 = dmdt ( v + .5 * DT * dvdt1, m + .5 * DT * dmdt1 );
60       double dhdt2 = dhdt ( v + .5 * DT * dvdt1, h + .5 * DT * dhdt1 );
61       double dndt2 = dndt ( v + .5 * DT * dvdt1, n + .5 * DT * dndt1 );
62       double dvdt2 = dvdt ( v + .5 * DT * dvdt1, m + .5 * DT * dmdt1, h + .5 * DT * dhdt1, n + .5 *
           DT * dndt1, i_ext );
63
64       double dmdt3 = dmdt ( v + .5 * DT * dvdt2, m + .5 * DT * dmdt2 );
65       double dhdt3 = dhdt ( v + .5 * DT * dvdt2, h + .5 * DT * dhdt2 );
66       double dndt3 = dndt ( v + .5 * DT * dvdt2, n + .5 * DT * dndt2 );
67       double dvdt3 = dvdt ( v + .5 * DT * dvdt2, m + .5 * DT * dmdt2, h + .5 * DT * dhdt2, n + .5 *
           DT * dndt2, i_ext );
68
69       double dmdt4 = dmdt ( v + DT * dvdt3, m + DT * dmdt3 );
70       double dhdt4 = dhdt ( v + DT * dvdt3, h + DT * dhdt3 );
71       double dndt4 = dndt ( v + DT * dvdt3, n + DT * dndt3 );
72       double dvdt4 = dvdt ( v + DT * dvdt3, m + DT * dmdt3, h + DT * dhdt3, n + DT * dndt3, i_ext )
          ;
73
74       m += DT * ( dmdt1 + 2 * dmdt2 + 2 * dmdt3 + dmdt4 ) / 6.;
75       h += DT * ( dhdt1 + 2 * dhdt2 + 2 * dhdt3 + dhdt4 ) / 6.;
76       n += DT * ( dndt1 + 2 * dndt2 + 2 * dndt3 + dndt4 ) / 6.;
77       v += DT * ( dvdt1 + 2 * dvdt2 + 2 * dvdt3 + dvdt4 ) / 6.;
78     }
79   }
```

Lines 6–17 of the code are for setting the constant. The time tick width μs (DT) (= 0.01 ms) and the calculation is done for T = 1000 ms (T). Thus, the number of iterations of this loop NT (NT) is $NT = T/\Delta t$. Lines 19–24 are functions of the gate variables $\alpha_x(V)$ and $\beta_x(V)$, lines 26–31 are formulas for the gate variables m, h, and n, and lines 33–39 are gate variable update formulas. The main body starts from line 41. Lines 43–46 set the initial values of the membrane potential and gate variables; line 48 sets the external current to $I_{\text{ext}} = 9.0\ \mu\text{A/cm}^2$; line 50 starts the loop for the time step, where variable is the loop counter that counts up to $NT = 100000$ [*3]. In lines 51 and 52, the time is set and the values of the membrane potential and gate variables are displayed. In lines 54–77, the values of the membrane potential and gate variables are updated using the Runge-Kutta method (Section 2.4). Here, constants set by `#define` should be written in upper cases, whereas variables in lower cases for readability. It is recommended that such notation be standardized to some extent.

As a note, in the original HH model, the resting potential is set to 0 mV, but in a normal membrane potential recording, the extracellular potential is set to 0 mV. In that case, all potential parameters are shifted by about −65 mV. To show this, the constant `E_REST` is set to −65 mV in line 6, and the entire membrane potential, including the value of the reversal potential, is shifted.

Let's compile and execute the code, output the calculation results to a file, and display the results in a graph. First, let's compile [*4].

```
node00:~/snsbook/code/part1/hh$ make hh
gcc -O3 -std=gnu11 -Wall -c hh.c
gcc -O3 -std=gnu11 -Wall -o hh hh.o -lm
```

`-O3` is the option for the most powerful optimization, `-std=gnu11` is the option to compile with GNU extention to the C11 specification, `-Wall` is the option to print all warning during compilation. When succeeded, an executable file `hh` is generated. If you run `hh` directly, it will display a lot of numbers on the screen, so you should redirect them to a file for output.

```
node00:~/snsbook/code/part1/hh$ ./hh > hh.dat
```

[*3]The reason for not calculating the 3 time loop as floating point is that counting in floating point may malfunction due to rounding errors (see section 2.5).

[*4]For more information about `make`, see the column "Let's write Makefile!".

After outputting the calculation results to a file, let's display them with some kind of graphing application. In this document, we use `gnuplot` (Section A.2), which is widely used on Windows, Mac, and Linux.

```
node00:~/snsbook/code/part1/hh$ gnuplot

        G N U P L O T
:
Terminal type set to 'x11'
gnuplot> plot 'hh.dat' with lines
```

The result should be as shown in Figure 3.1. You can see that the membrane potential rises and falls rapidly. This is the firing of spikes. If we continue to apply external current as in this simulation,

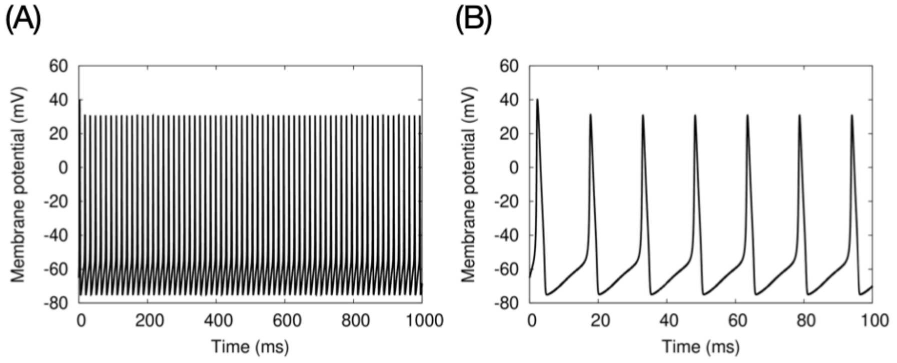


Fig. 3.1 Numerical simulation results of the HH model. (A) Calculation results for 1000 ms. (B) Magnified view of the first 100 ms.

the neuron will repeatedly and continuously fire spikes.

Let's take a closer look at the mechanism of spike firing. In the steady state, when a current is applied for a very short time, 1 ms, a single spike is fired (Figure 3.2. As the membrane is depolarized

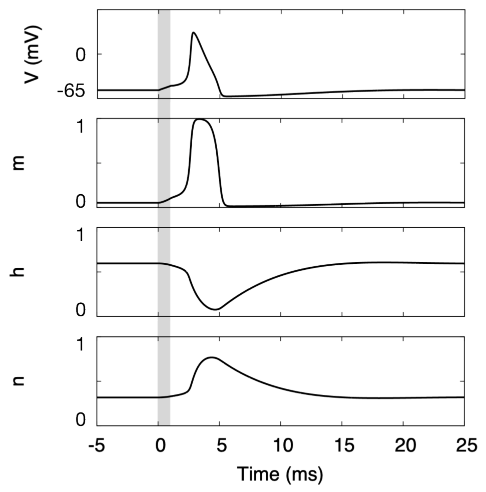


Fig. 3.2 Mechanism of spike firing. Gray shaded area indicates the time of stimulation.

by the current injection, the value of the gate variable m of the Na^+ channel increases. Then, more Na^+ ionic currents are generated, and the membrane is further depolarized. This positive feedback causes the membrane potential to 100 mV rise rapidly to a certain degree. When the value of the membrane potential becomes sufficiently high, the value of the gate variable h becomes 0, so the Na^+ current of the generated ions stops, and the membrane potential falls. This causes the membrane potential to rise and fall rapidly, and spikes are fired. In parallel, the K^+ ion current is generated on a time scale comparable to the change in the value of h. The current hyperpolarizes the membrane below the resting potential. This period is called the **refractory period**, during which the Na^+ channel is in an inactive state and cannot fire another spike. Eventually, the m, h, n value returns to a steady state, and the next spike is ready to be fired.

Thus, spikes are generated by the very fast dynamics of Na^+ the channel and K^+ channel conductance. Note that the dynamics of the rest of the membrane potential is a relatively slow dynamics subject to the time constant $\tau = RC = C/\overline{g}_{\mathrm{leak}}$ where R is the membrane resistance and is the reciprocal of $\overline{g}_{\mathrm{leak}}$.

3.1.3 Calculation of firing rate

The above simulation was set with $I_{\mathrm{ext}} = 9.0\ \mu\mathrm{A/cm}^2$ as the external current, but how would the spike number change if the current intensity was changed?

The number of spikes fired per second is called the **firing rate**, which will be explained in more detail in Section 3.6.1. The unit is spikes/s or Hz. In the case of the HH model, Figure 3.3 shows the firing rate calculated for a constant external current input for one second, while varying the current intensity. The relationship between the intensity of the current (I) and the firing rate (F) is called the **I-F curve**. Here, spike firing is detected as follows For an appropriate constant θ.

$$V(t) > \theta \text{ and } V(t - \Delta t) < \theta \Longleftrightarrow \text{Spikt was fired at time } t \tag{3.6}$$

θ is typically set to 0 mV (when the resting potential is -65 mV). The condition on the left-hand side means that the membrane potential crosses the threshold θ from bottom to top at a certain time t. The time when the membrane potential crosses the threshold is defined as the spike firing time.

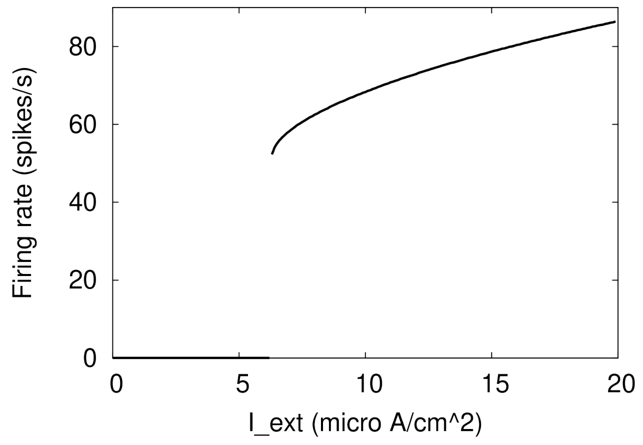


Fig. 3.3 I-F curve of HH model.

The HH model with this parameter fires about 52 spikes/s when the intensity of the external current is $I_{\mathrm{ext}} = 6.3\ \mu\mathrm{A/cm}^2$, but does not fire any spikes when $I_{\mathrm{ext}} = 6.2\ \mu\mathrm{A/cm}^2$. Thus, a neuron that suddenly fires a high frequency of spikes when the external current exceeds a certain value is called a Type II (or Class II) neuron (Hodgkin and Huxley, 1952; Izhikevich, 2010)。

On the other hand, neurons in the cerebral cortex have the property of continuously increasing their firing frequency in proportion to the intensity of the incoming external current (Erisir et al., 1999). Such neurons are referred to as Type I (or Class I) neurons (Hodgkin and Huxley, 1952). In addition, neurons in so-called neural networks, which are widely used in deep learning, are also assumed to continuously increase their activity level in proportion to the input intensity. How can such a property be obtained?

3.1.4 Extending the HH model

The above extensions can be obtained by adding new currents to the HH model. To distinguish this from the original HH model, we call it the **HH-type** model.

Adaptation of firing rate

In the HH model based on the giant axon of the giant squid, the spike firing interval was constant for a steady current input. However, in real neurons, the firing interval becomes wider and wider, and the firing rate decreases. This is called the **adaptation** of firing rate.

Adaptation of the firing rate can be realized by adding a current, the **after-hyperpolarizing current** $I_{\mathrm{ahp}}(t)$, to the HH model [*5](Treves, 1993).

$$I_{\mathrm{ahp}}(t) = -\overline{g}_{\mathrm{ahp}} a(t) \left(V(t) - E_{\mathrm{K}} \right) \tag{3.7}$$

where $\overline{g}_{\mathrm{ahp}}$ is the maximum conductance value and $a(t)$ is the gate variable. $a(t)$ is updated by the following equation

$$\tau_{\mathrm{ahp}} \frac{da}{dt} = -a(t) + S(t) \tag{3.8}$$

Here, τ_{ahp} is a time constant that is large enough to allow for gradual changes in firing rate over multiple firings, and $S(t)$ is set to 1 if you fire a spike at time t and 0 otherwise. In other words, the variable a takes the cumulative value of the spikes fired up to that point. This means that if more spikes are fired, a proportional hyperpolarizing current will flow, reducing the firing rate.

`code/part1/hh/sfa.c` [*6] is the code for the HH model (List 3.1) with $I_{\mathrm{ahp}}(t)$. Figure 3.4 shows the waveforms of the membrane potential during the first and last 100 ms for a steady current input of 1 s. In the first 100 ms, 10 spikes are fired, but in the last 100 ms, only 7 spikes are fired. In terms of firing rate, this means that the rate has decreased from 100 spikes/s to 70 spikes/s.

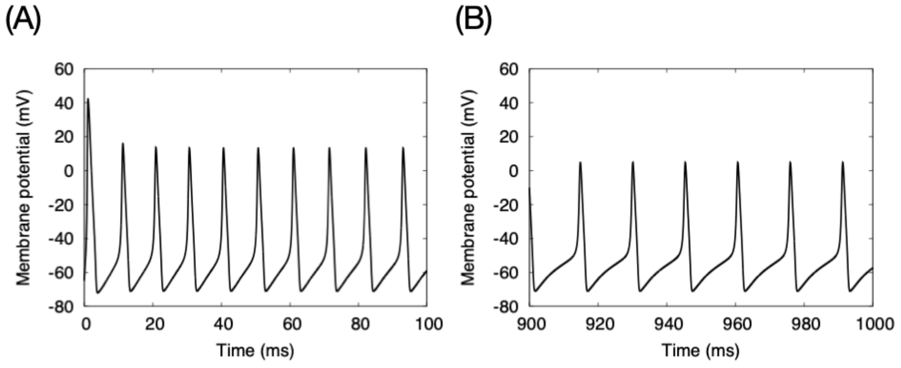


Fig. 3.4 Adaptation of firing rate. (A) Membrane potential during the first 100 ms. (B) Membrane potential during the last 100 ms. The parameters are $\overline{g}_{\mathrm{ahp}} = 1400$ mS/cm^2, $\tau_{\mathrm{ahp}} = 200$ ms, $I_{\mathrm{ext}} = 40$ μA/cm^2.

[*5]There are many other ways to do this, but here are the simplest ones.

[*6]Spike Frequency Adaptation, SFA.

It is also known that such adaptation is L-caused by a type of calcium channel involving the intracellular calcium ion Ca^{2+} (Tanabe et al., 1998). In fact, the above gating variables can be a regarded as intracellular Ca^{2+} concentrations.

Model of a Type I neuron

One way to obtain Type I neurons is to apply a transient current of K^+ ions, called the I_A-current, to Type II neurons (Connor and Stevens, 1971; Connor et al., 1977). The I_A-current hyperpolarizes the membrane with a relatively short time constant ($\tau_A \approx 10$ ms) and delays spike firing. Under weak steady-state currents, spikes can be fired only after the effect of the I_A-current is sufficiently weak, resulting in a lower firing rate.

As an example of a Type I neuron, Figure 3.5 shows a test implementation of the Connor-Stevens model introduced in the literature (Connor et al., 1977). Figure 3.5A shows the case where the external current is set to $I_{\rm ext} = 9.0\ \mu$A/cm^2, as in `hh.c`. Although the firing rate is very low, the waveform is similar to that of the HH model. When the external current is gradually weakened, the normal HH model fires at more than 50 spikes/s, but suddenly stops. In this model, on the other hand, when the external current is lowered to $I_{\rm ext} = 8.61\mu$A/cm^2, the rate decreased to 6 spikes/s (Fig. 3.5B). the I-F curve is plotted in Fig. 3.6, and it can be seen that the firing frequency increases more smoothly than in the HH model.

The implementation of the Connor-Stevens model has different parameters from the HH model, but it is essentially just adding one current to the HH model, so I will not explain the code. The code for the Connor-Stevens model can be found at `code/part1/hh/ia.c`.

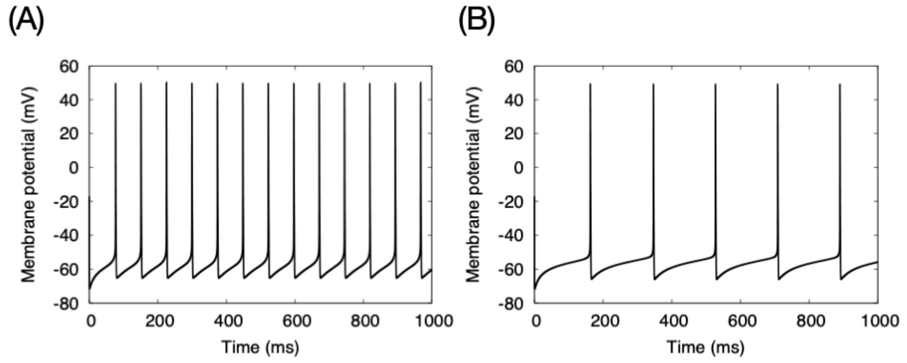


Fig. 3.5 Firing pattern of a Type I neuron. (A) $I_{\rm ext} = 9.0\mu$A/cm^2. (B) $I_{\rm ext} = 8.61\mu$A/cm^2.

3.1.5 On the step width of numerical simulations

By the way, when we try to solve the HH model using the Euler method, we need to reduce Δt to about 10 μs. In fact, at 10 times 100μs, the calculation diverges in the middle. This is because the spike firing part, especially the part where the membrane potential rises rapidly, has very fast dynamics and requires even smaller time resolution than that. In contrast, the dynamics of the membrane potential outside the spike firing region is much slower. In fact, the time constant of the membrane potential in the HH model is $C/\overline{g}_{\rm leak} = 3.3$ ms and $\Delta t = 100\mu$s is sufficient.

If we assume that brain function arises from the dynamics of a network of many neurons exchanging spikes, then we can ignore the precise temporal dynamics of individual neurons firing spikes. By doing so, we can increase Δt by a factor of 10–100, and thus speed up the computation time by a factor of 10–100. Based on this idea, there is a simplified version of the HH model, the **leaky integrate-and-fire model** described in Section 3.2.

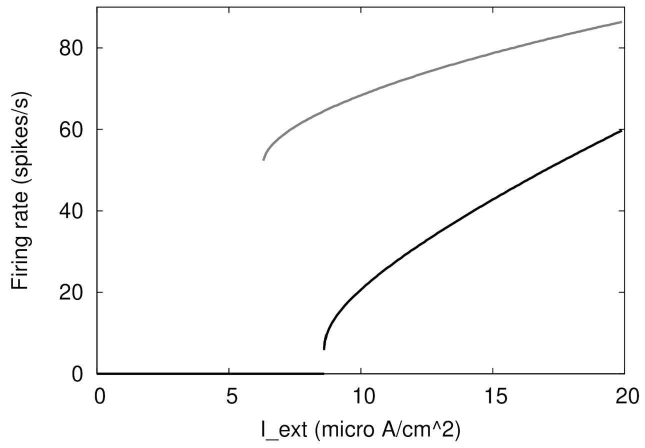


Fig. 3.6 I-F curve of a Type I neuron. Gray is the curve for a Type II neuron (HH model).

3.1.6 Staggered time step *

In the calculation of the HH model, the gate variables of the ion channel (m, h, n) and the membrane potential variable (V) were updated simultaneously. In other words, Δm, Δh, Δn, and ΔV are calculated in advance and then updated together.

$$\begin{aligned} m(t+\Delta t) &= m(t) + \Delta m \\ h(t+\Delta t) &= h(t) + \Delta h \\ n(t+\Delta t) &= n(t) + \Delta n \\ V(t+\Delta t) &= V(t) + \Delta V \end{aligned}$$

This is a correct method in terms of numerical calculation, and there is no problem at all. In particular, if you use a method like Euler's method that does not require recalculation, it does not matter how you write it. However, if we use a higher-order numerical solution method, such as Runge-Kutta method, we have to repeat the calculation of these four variables four times at the same time. In this case, the calculation of V includes the calculation of h^3 and n^4, which makes the calculation heavy, and more importantly, the calculation of nonlinear equations. Furthermore, since the code writer wants to make the code as modular as possible, the ion channel calculation and the membrane potential calculation should be handled separately.

Therefore, we introduce the method of **staggered time step** [*7] (Mascagni and Sherman, 1998; Carnevale and Hines, 2006). The calculation of the ion channel is conventionally done starting from $t = 0$ at $t = \Delta t$, $2\Delta t$, $\cdots$, but this method starts by shifting the initial state by $-\Delta t/2$ and calculates at $t = \Delta t/2$, $3\Delta t/2$, $5\Delta t/2$, $\cdots$. The important conditions are as follows:

- Always start with a steady state as the initial condition ($x(0) = x(-\Delta t/2) = x(-\Delta t)$, where x is the ion channel or membrane potential variable)
- The ion channel calculation uses the value of the membrane potential before the $\Delta t/2$ step
- The membrane potential is calculated in the same way, using the value of the ion channel before the $\Delta t/2$ step

In this way, the flow of computation is changed from

[*7] It is generally called the Leapfrog method. This is because, literally, the two variables are calculated by shifting the time of each other like a frog jumping.

1. Calculation of Δm, Δh, Δn, and ΔV
2. Update m, h, n, and V

to

1. Calculation of Δm, Δh, and Δn
2. Update m, h, and n
3. Calculation of ΔV
4. Update V

The more ion channels there are, the more useful this method becomes. Most importantly, in this way, the ion channel calculation can be regarded as linear and the membrane potential calculation as linear [*8].

3.2 Simulation of leaky integrate-and-fire models

3.2.1 Leaky integrate-and-fire model

Since the HH model is a model that properly calculates the process of spike firing, we needed to use four variables per neuron (V, n, m, h) and a relatively small Δt (10 μs). When simulating a large number of neurons at the same time, using a large number of variables consumes a large amount of memory, and a small Δt means that the calculation takes a long time.

Many physiological experiments have shown that the waveform of the spike itself is not informative, but the number of spikes and the timing of the firing is informative. Based on this assumption, we can omit the calculation of spike generation, reduce the number of variables, and increase Δt. Based on such an idea, simpler neuron models have been proposed. The simplest one is the **leaky integrate-and-fire (LIF) model**, which has only one variable as a parameter.

The LIF model is described by the following equation:

$$\tau \frac{dv}{dt} = -\left(v(t) - V_{\text{rest}}\right) + RI_{\text{ext}}(t) \tag{3.9}$$

$$v(t) > \theta \Rightarrow S(t) = 1, v(t) \leftarrow V_{\text{reset}} \tag{3.10}$$

$$v(0) = V_{\text{init}} \tag{3.11}$$

where $v(t)$ (mV) is the membrane potential at time t, τ (ms) is the time constant, V_{rest} (mV) is the rest potential, R (MΩ) is the resistance of the membrane, $I_{\text{ext}}(t)$ (nA) is the external current at time t, θ (mV) is the **threshold** for spike firing, V_{reset} (mV) is the **reset potential**, V_{init} (mV) is the initial value of the membrane potential is the initial value of the membrane potential. Equation (3.9) describes the basic membrane potential dynamics. The membrane potential has an equilibrium point at V_{rest}, and asymptotes to $V_{\text{rest}} + RI_{\text{ext}}(t)$ with an external input $RI_{\text{ext}}(t)$ and a time constant τ. Equation (3.10) is the condition for spike firing. When the membrane potential exceeds the threshold, the spike is assumed to be fired at that time $(S(t) = 1)$ and the membrane potential is reset. Equation (3.11) gives the initial value of the membrane potential.

Note that the LIF model does not have the capacitance and conductance that existed in the HH model. multiplying both sides of equation (3.1) by $1/\overline{g}_{\text{leak}}$, we get equation (3.9) if we note that $R = 1/\overline{g}_{\text{leak}}$. where $\tau = RC$. A model that does not explicitly describe conductance is called a **current-based model** (a model that explicitly describes conductance is a **conductance-based model**).

Note that the parameters in the HH model are in units of per unit area (1/cm^2), but this is no longer the case in the LIF model. In the HH model, the exact units are retained because they are

[*8]Because the value of the membrane potential appears to be a constant from the equation of the ion channel and the value of the ion channel appears to be a constant from the equation of the membrane potential, respectively.

estimated from actual neurons, and when considering the spatial shape of the cell, as in the multi-compartment model (Section 3.7) described below, the surface area of the membrane is needed, so these units must be retained. The LIF model, on the other hand, is abstracted so that it does not have a shape, and can be thought of as pre-normalized.

In the LIF model, the spike generation mechanism that was the core of the HH model is replaced by a threshold and a reset potential (Equation (3.10)). Since the threshold and reset potential are constants, the only variable is the membrane potential. At the same time, the LIF model does not calculate the specific waveform of the spike, but simply provides the time at which the spike was fired. This means that Δt can be much larger. For example, a Δt of about 1 ms does not break the calculation, and the simulation can be done properly; solving the HH model using the Euler method would have required about 10 μs, so the calculation is about 100 times faster[*9] .

As a minor detail, since the LIF model only replaces the detailed dynamics of spike firing in the HH model with a threshold, capacitance and conductance may be described explicitly. Note that LIF model does not always mean a current-based model.

3.2.2 LIF model simulation

Now let's try to simulate a single neuron in the LIF model, which, unlike the HH model, does not need to compute the elaborate dynamics of spike generation, so we will use the Eulerian method and solve for $\Delta t = 1$ ms. All the code can be found in `code/part1/lif/`.

Listing 3.2 `lif.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
#include <stdbool.h>

#define TAU     (   20.0 ) // ms
#define V_REST  (  -65.0 ) // mV
#define V_RESET (  -65.0 ) // mV
#define THETA   (  -55.0 ) // mV
#define R_M     (    1.0 ) // MOhm
#define DT      (    1.0 ) // ms
#define T       ( 1000.0 ) // ms; unused
#define NT      ( 1000   ) // ( T / DT )
#define I_EXT   (   12.0 ) // nA

int main ( void )
{
  float v = V_REST;

  for ( int32_t nt = 0; nt < NT; nt++ ) {
    float t = DT * nt;
    printf ( "%f␣%f\n", t, v );

    v += DT * ( - ( v - V_REST ) + R_M * I_EXT ) / TAU;
    bool s = ( v > THETA );

    // Pretty-print spikes on membrane potentials. Note that spike time is not t but t + DT
    if ( s ) { printf ( "%f␣%f\n%f␣%f\n", t + DT, v, t + DT, 0. ); }

    v = s * V_RESET + ( ! s ) * v;
  }
}
```

[*9]Moreover, the number of time-consuming floating-point operations (required to calculate gate variables) is far fewer.

Compared to the HH model, the code is very simple. Compared to the HH model, the number of loop rotations is 1/100, and the number of floating point operations per loop is also 1/100 or less, so the total number of operations is 1/10000 or less [*10].

In this code, we simulate $T = 1000$ ms duration (`T`) with $\Delta t = 1$ ms (`DT`). The membrane resistance R is normalized to 1 MΩ (`R_M`), and the external current $I_{\mathrm{ext}} = 12$ nA (`I_EXT`) is given. The membrane potential, $v(t)$, is calculated from the initial value, $v(0) =$ `V_REST` $= -65$ mV, for each Δt successively, `NT` $=$ `T/DT` times.

The execution of the code starts at line 17, `main`. At line 19, the variable `v` is initialized to `V_REST` $= -65$ mV, and at line 21, we enter a loop with respect to time. At lines 22 and 23, we display the current value of the membrane potential with time. At line 25, we estimate the membrane potential at the next time, i.e., $t + \Delta t$. In line 25, the membrane potential at the next time, i.e., $t + \Delta t$, is estimated, and in line 26, if the threshold `THETA` is exceeded, a spike is fired. The spike is stored in the variable `s` as true or false [*11]. In line 29, if a spike is firing, we output a spike-like waveform so that it can be displayed; in line 31, we actually update the membrane potential; as a result of our calculations in line 25, if a spike is firing (i.e., `s` is true), we reset the membrane potential to `V_RESET`$= -65$ mV if a spike is fired (i.e., `s` is true), otherwise (i.e., `!s` is true), and if not (i.e. `!s` is true), the calculated value is the value of the membrane potential at the next time $t + \Delta t$.

In this way, the LIF model repeats the behavior of the membrane potential rising toward the threshold and resetting when the threshold is reached. Note that only the time when the spike was fired is available, not the waveform of the spike. Therefore, we can obtain a plausible looking waveform by artificially plotting the spike-like vertical bars. That is the role of line 29. Let's run this code as we did for the HH model and display the results.

```
node00:~/snsbook/code/part1/lif$ make lif
gcc -O3 -std=gnu11 -Wall -c lif.c
gcc -O3 -std=gnu11 -Wall -o lif lif.o -lm
node00:~/snsbook/code/part1/lif$ ./lif > lif.dat
node00:~/snsbook/code/part1/lif$ gnuplot
:
gnuplot> plot 'lif.dat' with lines
```

We will obtain a plot of membrane potential as in Figure 3.7. We can see that spikes are fired at regular intervals for 1s.

3.2.3 When to reset the membrane potential

In some textbooks (e.g. Trappenberg (2010)), the code for the LIF model is written as follows:

Listing 3.3 `lif_alt.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
#include <stdbool.h>

#define TAU     (   20.0 ) // ms
#define V_REST  (  -65.0 ) // mV
#define V_RESET (  -65.0 ) // mV
#define THETA   (  -55.0 ) // mV
```

[*10] When the exp function is set to 10 operations and the total number of operations is simply counted. However, the number of operations of the exp function differs depending on the computer, and labor saving occurs when the same operation is repeated. Furthermore, depending on the computer, operations in single precision (float) are 2–4 times faster than those in double precision (double), so the difference in calculation time opens up further.

[*11] `#include<stdbool.h>`, so it can use boolean values, but the entities are 1 or 0.

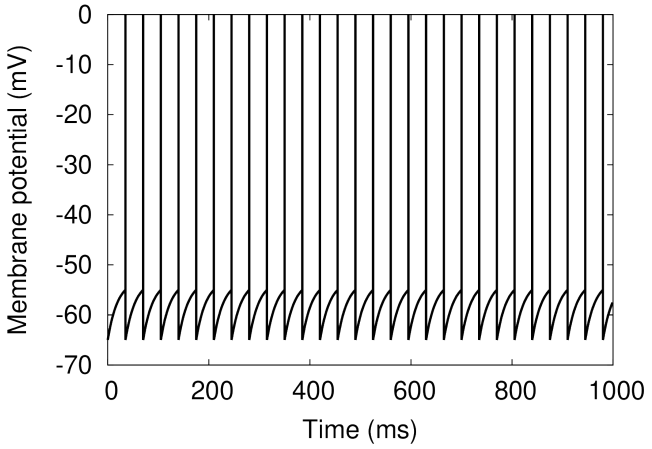


Fig. 3.7 Plot of the membrane potential of a single neuron, where the vertical bar extending to 0 mV represents the time of spike firing.

```
#define R_M     (    1.0 ) // MOhm
#define DT      (    1.0 ) // ms
#define T       ( 1000.0 ) // ms; unused
#define NT      ( 1000   ) // ( T / DT )
#define I_EXT   (   12.0 ) // nA

int main ( void )
{
  float v = V_REST;

  for ( int32_t nt = 0; nt < NT; nt++ ) {
    float t = DT * nt;
    printf ( "%f␣%f\n", t, v );

    bool s = ( v > THETA );
    float dv = DT * ( - ( v - V_REST ) + R_M * I_EXT ) / TAU;

    v = s * V_RESET + ( ! s ) * ( v + dv );
  }
}
```

The code says that if the membrane potential is above the threshold θ at time t, then reset the membrane potential at the next time $t + \Delta t$. The diagram is shown in Figure 3.8A. Since the threshold is not exceeded at time $t - \Delta t$ and is exceeded for the first time at time t, we assume that the spike was fired at this time and reset the membrane potential at the next time $t + \Delta t$. From the way it is discretized, this seems to be correct.

On the other hand, the code in this book is as shown in Figure 3.8B. At time t, we estimate the membrane potential at the next time $t + \Delta t$, and if it exceeds the threshold at the next time, we assume that a spike will be fired at that time $(t + \Delta t)$, and reset the membrane potential without letting it exceed the threshold. The reason for doing this is that the process of the membrane potential crossing the threshold, firing a spike, and resetting is completed within the time width Δt, and the membrane potential has already been reset at time $t + \Delta t$. This seems to be correct.

Both seem to be correct, but the time at which the membrane potential is reset shifts by Δt. This is ultimately a side effect of the discretization with respect to time, because if Δt were closer to 0, the two would agree. This is a limitation of the LIF model, which does not explicitly account for firing, and if the effect of firing times on the order of less than 1 ms is important to you, make Δt small enough or use a different model.

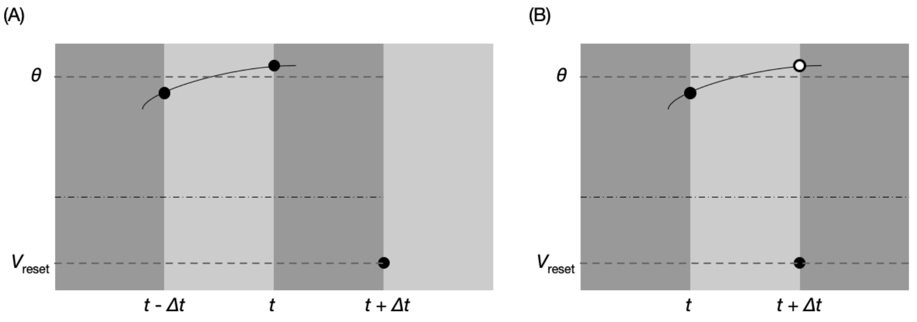


Fig. 3.8 Timing of resetting the membrane potential in the LIF model. (A) Some textbook methods. (B) Method in this book.

3.2.4 Introduction of refractory period *

The HH model enters a refractory period immediately after the spike is fired due to the (Figure 3.2 around 5—10 ms) inactivation of the Na^+ channel or the K^+ activated state of the channel. while the LIF model has no refractory period because it does not have a gating mechanism. Therefore, in order to achieve a refractory period in the LIF model, it is necessary to introduce another mechanism such as the following:

- Method of adding posterior hyperpolarizing current (I_{ahp}) ahp(Section 3.1.4)
- How to lift the threshold instantaneously
- Method to force the value of the membrane potential to be fixed at V_{reset}

The first method applies a current with a short time constant corresponding to the length of the refractory period. The last method is to introduce a counter explicitly, and the difference from `lif.c` is shown in Listing 3.4 below [*12].

Listing 3.4 `diff -u lif.c lif_refr.c`

```
--- lif.c       2021-04-26 22:39:04.000000000 +0900
+++ lif_refr.c  2021-04-26 22:38:43.000000000 +0900
@@ -13,10 +13,13 @@
 #define T       ( 1000.0 ) // ms; unused
 #define NT      ( 1000   ) // ( T / DT )
 #define I_EXT   (   12.0 ) // nA
+#define T_REFR  (    5.0 ) // ms; unused
+#define NT_REFR (    5   ) // ( T_REFR / DT )

 int main ( void )
 {
   float v = V_REST;
+  int32_t refr = 0; // counter for refractory period

   for ( int32_t nt = 0; nt < NT; nt++ ) {
     float t = DT * nt;
@@ -28,6 +31,7 @@
     // Pretty-print spikes on membrane potentials. Note that spike time is not t but t + DT
     if ( s ) { printf ( "%f␣%f\n%f␣%f\n", t + DT, v, t + DT, 0. ); }

-    v = s * V_RESET + ( ! s ) * v;
+    refr = s * ( NT_REFR ) + ( ! s ) * ( refr - 1 ); // set counter
```

[*12] `diff` is a command to output the difference between two files, where a leading `+` indicates a line that has been added, and a leading `-` indicates a line that has been deleted.

```
23 +     v = ( refr > 0 ) ? V_RESET : v;
24     }
25   }
```

First, we set the length of the refractory period to `T_REFR` (5 ms in this case; we do not use this constant itself) (line 7), and `NT_REFR` to the number of steps converted to simulation steps by dividing by Δt (line 8). Introduce and initialize a counter `refr` for the refractory period (line 13), and set the value of the counter to `NT_REFR` if a spike is fired, otherwise set the value to -1 (line 22). While the counter value is positive (i.e. during the refractory period), the membrane potential is fixed at `V_RESET` (line 23). Last sentence:

```
23 +     v = ( refr > 0 ) ? V_RESET : v;
```

is a notation called **ternary operator**, which returns `V_RESET` or `v` depending on whether the condition `( refr > 0 )` is true or false. This is more concise than using if statements, so use it aggressively.

This allows us to introduce a refractory period into the LIF model as well. However, if the dynamics during the refractory period may play an important role, it is more natural to use a HH-type model instead of such an artificial implementation.

3.3 Other neuron models

3.3.1 Izhikevich model

The HH model was a four-variable model that used the membrane potential $V(t)$ and $m(t)$, $h(t)$, and $n(t)$ for spike generation, while the LIF model was a one-variable model consisting of only the membrane potential $v(t)$. The LIF model was a one-variable model consisting only of the membrane potential $v(t)$. In both models, new currents had to be added to make the behavior more complex, and the number of required variables increased accordingly. The more variables, the more complicated the calculation becomes, and the more calculation time and memory are required.

Eugine Izhikevich, who mathematically analyzed the behavior of a single neuron, developed a neuron model consisting of two variables (Izhikevich, 2003, 2010). This model, called the **Izhikevich model**, is described by the following equation:

$$\begin{aligned} C\frac{dV}{dt} &= k(V(t) - E_{\mathrm{leak}})(V(t) - E_{\mathrm{t}}) - u(t) + I_{\mathrm{ext}}(t) \\ \frac{du}{dt} &= a\left(b(V(t) - E_{\mathrm{leak}}) - u(t)\right) \end{aligned} \tag{3.12}$$

where C is the capacitance, $V(t)$ is the membrane potential at time t, E_{leak} is the resting potential, E_{t} is the threshold potential that represents the upper limit of the membrane potential, $u(t)$ is an internal parameter called the return current, and k, a, and b are constants. When the membrane potential $V(t)$ exceeds the threshold θ, the neuron is assumed to fire a spike, and $V(t)$ and $u(t)$ are set to c and $u(t) + d$, respectively. where the constant c is the reset potential and d is the intensity of the hyperpolarization. Although there are only two variables, $V(t)$ and $u(t)$, by varying the constants a–d, the model can reproduce the various behaviors exhibited by a real neuron (Figure 3.9).

3.3.2 Poisson spike

In spiking network simulation, there are many situations where we want to generate spike trains without assuming a specific neuron model. For example, we may want to give random spikes to all

[*1] `https://www.izhikevich.org/publications/whichmod.htm`

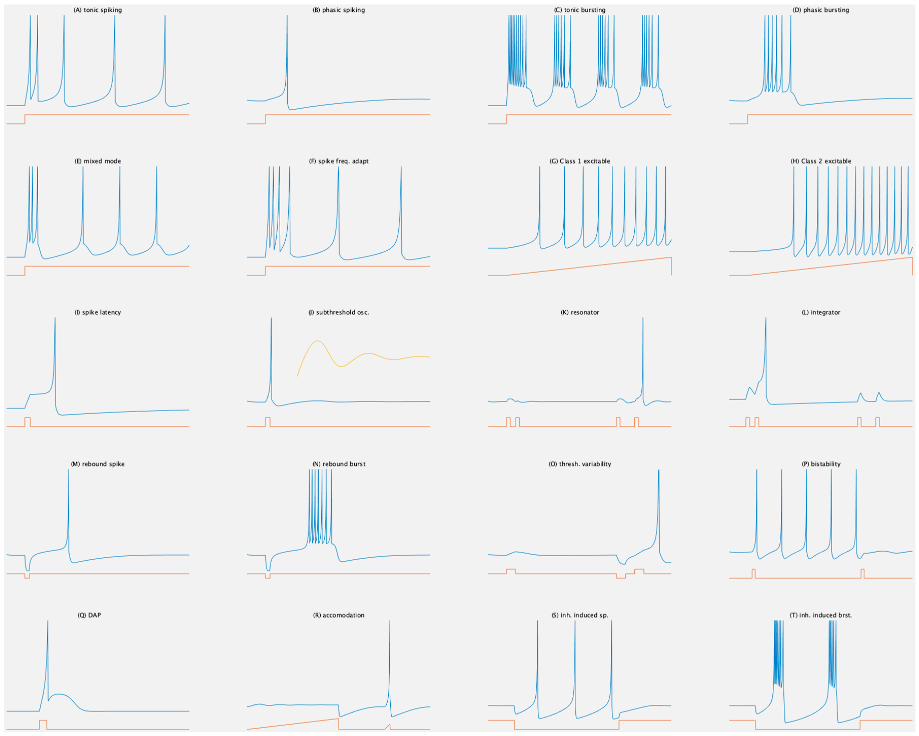


Fig. 3.9 Various firing patterns using the Izhikevich model, created by running the Matlab script on Izhikevich's home page[*1].

neurons as background noise [*13], or we may want to generate spikes based on the firing frequency of external stimuli. In such cases, **Poisson spikes** are used.

The method for generating Poisson spikes is quite simple. Let Δt (`DT`) be the time increment of the simulation. First, the method to generate random spikes with a constant firing frequency f (spikes/s) (`f`) is as follows:

```
1 for ( int32_t nt = 0; nt < NT; nt++ ) {
2   double t = nt * DT;
3   double r = sfmt_genrand_real2 ( &rng );
4   bool s = ( r < f * DT );
5 }
```

The first line is the loop on simulation steps. The number of iterations is calculated as $NT = T/\Delta t$ ($=$ `NT`), where T is the total simulation time. The second line converts the number of steps to time t (to be used later), the third line generates a random number $[0, 1)$, and the fourth line calculates the spike firing. The fourth line is the computation of spike firing, where `s` (true or false) indicates the presence or absence of spike firing. The fourth line is the spike firing calculation, where s (true or false) indicates whether the spike will be fired or not. Simply put, each step is a random number that determines whether the spike will be fired or not.

To simplify things, let's start with a 1 s simulation with a time interval of 1 ms and a firing frequency of 100 spikes/s. In terms of s, $T = 1$, $\Delta t = 0.001$, $NT = 1000$, and $f = 100$, and in terms of ms, $T = 1000$, $\Delta t = 1$, $NT = 1000$, $f = 0.1$. The reason why the value of f is 0.1 in the latter case is that 100 shots per second means $100/1000 = 0.1$ shots per 1 ms. If we proceed with the latter, we will enter the 4th line of calculation by shaking a random number at each step of the loop, and

[*13]Some cells in the cerebral cortex fire at extremely low frequency and random timing when they are not engaged in any meaningful activity, or when they are not actively generating firings. This kind of seemingly meaningless activity is sometimes called background noise, because it is like background noise.

since the value of $f\Delta t$ is now 0.1, s = true with a probability of 1/10 (i.e., it will fire a spike). Since each loop fires a spike with a probability of 1/10 and the loop is repeated $NT = 1000$ times, it makes sense that on average $0.1 \times 1000 = 100$ spikes will be fired. If the firing frequency is 50 spikes/s based on ms, then $f\Delta t = 0.05 \times 1 = 0.05$, so if the loop is repeated 1000 times, on average 50 spikes will be fired. Alternatively, if $\Delta t = 5$ ms, then $NT = 1000/5 = 200$ and $f\Delta t = 0.1 \times 5 = 0.5$, so if the loop is repeated 200 times and spikes are fired with a probability of 0.5 each time, then the average The target number of spikes is 100.

Thus, a Poisson spike is a random number that is rolled each time in a time loop to determine whether to fire or not. The spike firing will be independent in each trial. The reason why this is called a Poisson spike is that such a discrete stochastic process that becomes independent each time is called a Poisson process.

In the above example, the firing frequency is constant at f, but it can be calculated for the time-varying case $f(t)$ in the same way as follows:

```
for ( int32_t nt = 0; nt < NT; nt++ ) {
  double t = nt * DT;
  double r = sfmt_genrand_real2 ( &rng );
  bool s = ( r < f ( t ) * DT );
}
```

3.4 Simulation of synapses

We have already mentioned that the change in membrane potential due to spike propagation can be represented by adding the synaptic current (Equation (1.3)), which corresponds to a change in the conductance value $g_{\text{syn}}(t)$. The calculation of a synapse corresponds to varying the conductance value $g_{\text{syn}}(t)$, where one pre-side neuron is coupled to one post-side neuron. The basic model of a synapse is called an **exponentially-decaying synapse**, where the conductance $g_{\text{syn}}(t)$ is calculated by the following equation:

$$g_{\text{syn}}(t) = \overline{g}_{\text{syn}} \sum_{f \in S(t)} \exp\left(-\frac{t - t^{(f)}}{\tau}\right) \Theta\left(t - t^{(f)}\right) \tag{3.13}$$

Here, $\overline{g}_{\text{syn}}$ is the maximum conductance value per spike, which is a constant determined by, for example, the amount of neurotransmitter released from the pre-side and the density of receptors on the post-side. The pre-side neuron fires a sequence of spikes $S(t)$, where $t^{(f)}$ is the firing time of the spike f, and τ is the time constant that represents the decay of the conductance value per spike. Specifically, τ represents the time when the conductance value decays $1/e$. $\Theta(x)$ is the Heaviside step function, and $\Theta(x) = 1$ $(x \geq 0)$ or 0 $(x < 0)$ [*14]。

Figure 3.10 shows how conductance changes at an exponentially decaying synapse. When a single spike arrives from the pre-side neuron, the conductance rises instantaneously and then decays back to zero according to the time constant τ. The product of the conductance and the difference of the membrane potential from the reversal potential generates the synaptic current (Equation (1.3)). If several spikes arrive at the same time or at very short time intervals, the value of the synaptic current will be their superposition.

Next, we will introduce a more complex model of synapses. In the case of an exponentially decaying synapse, the current is generated instantaneously upon arrival of the spike, but in reality the current is generated gradually after arrival. We can introduce such an amplification process. In this paper, we will look at the difference between the two types of exponential functions.

[*14]For the Heaviside step function, the value at $x = 0$ is undefined, but values of 0, 1/2, and 1 are often used. In this document, it is assumed to be 1.

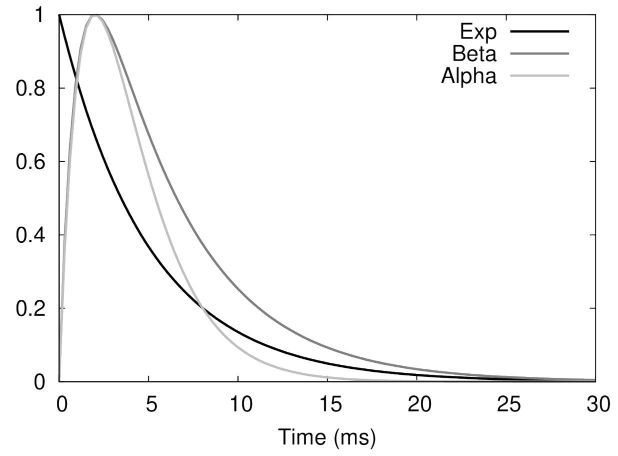


Fig. 3.10 Time evolution of synaptic conductance. The shapes of the exponential (Equation (3.13)), beta (Beta, Equation (3.14)), and alpha (Alpha, Equation (3.15)) functions are plotted as black, gray, and light gray lines, respectively. The parameters are $\tau = \tau_{\text{decay}} = 5$ ms, $\tau_{\text{rise}} = 1$ ms, $\tau'' = \tau' \ln(\tau_{\text{decay}}/\tau_{\text{rise}}) \approx 2.0$ ms, respectively. The horizontal axis is time (ms).

$$g_{\text{syn}}(t) = \overline{g}_{\text{syn}} B \sum_{f \in S(t)} \left(\exp\left(-\frac{t - t^{(f)}}{\tau_{\text{decay}}}\right) - \exp\left(-\frac{t - t^{(f)}}{\tau_{\text{rise}}}\right) \right) \Theta\left(t - t^{(f)}\right) \tag{3.14}$$

Here, τ_{rise} and τ_{decay} are the time constants of amplification and decay, respectively, and $\tau_{\text{rise}} < \tau_{\text{decay}}$. Since it is a subtraction of exponential functions with different time constants, the value starts at 0 at time t = 0, reaches a maximum value at $t = \tau' \ln(\tau_{\text{decay}}/\tau_{\text{rise}})$, and then gradually decays. And, $B = \left((\tau_{\text{rise}}/\tau_{\text{decay}})^{\tau'/\tau_{\text{decay}}} - (\tau_{\text{rise}}/\tau_{\text{decay}})^{\tau'/\tau_{\text{rise}}} \right)^{-1}$ is a constant that normalizes the maximum value at 1. Equation (3.14) has the name **β-function**. When we set $\tau_{\text{rise}} = \tau_{\text{decay}} = \tau''$, we obtain [*15]

$$g_{\text{syn}}(t) = \overline{g}_{\text{syn}} \sum_{f \in S(t)} \frac{t}{\tau''} \exp\left(1 - \frac{t - t^{(f)}}{\tau''}\right) \Theta\left(t - t^{(f)}\right) \tag{3.15}$$

Equation (3.15) is called the **α-function**, which takes its maximum value at $t = \tau''$.

3.4.1 Numerical method for synapses

Now, when we calculate the synaptic conductance, we need to sum (or more precisely, convolute with the exponential function) all the spikes we have received in the past. The simplest way to do this is to keep all the previous spikes and recalculate them each time at each step, but this is very inefficient, so we want to do it in a smarter way.

We will use the following equation, since Srinivasan and Chiel (1993) shows that the same calculation can be done. First, for an ordinary exponentially decaying synapse, the conductance value $g_{\text{syn}}(t)$ is calculated as

$$g_{\text{syn}}(t) = \overline{g}_{\text{syn}} \text{Sum}_1(t) \tag{3.16}$$

where

$$\text{Sum}_1(t + \Delta t) = \exp\left(-\frac{\Delta t}{\tau}\right) \text{Sum}_1(t) + S(t) \tag{3.17}$$

[*15] In equation (3.14), we set $\tau_{\text{rise}} = \tau_{\text{decay}} = \tau''$ and use L'Hopital's theorem.

and $\mathrm{Sum}_1(0) = 0$, $S(t)$ is a function that takes the values 1 and 0 depending on whether a spike has occurred at time t. Since the *beta* function is a subtraction of the exponential functions, we assume $\mathrm{Sum}_2(0) = 0$ and set

$$g_{\mathrm{syn}}(t) = \overline{g}_{\mathrm{syn}} B \left(\mathrm{Sum}_1(t) - \mathrm{Sum}_2(t)\right) \tag{3.18}$$

$$\begin{aligned} \mathrm{Sum}_1(t + \Delta t) &= \exp\left(-\frac{\Delta t}{\tau_{\mathrm{decay}}}\right) \mathrm{Sum}_1(t) + S(t) \\ \mathrm{Sum}_2(t + \Delta t) &= \exp\left(-\frac{\Delta t}{\tau_{\mathrm{rise}}}\right) \mathrm{Sum}_2(t) + S(t) \end{aligned} \tag{3.19}$$

In addition, a method to calculate the α function is introduced in the paper. As a generalization of these methods, a method called the Matrix Exponential method has also been proposed (Rotter and Diesmann, 1999).

Another method is to write the conductance equation as a differential equation from the beginning (Trappenberg, 2010). Differentiate both sides of equation (3.13) yields

$$\frac{dg_{\mathrm{syn}}}{dt} = -\frac{1}{\tau} g_{\mathrm{syn}}(t) + \overline{g}_{\mathrm{syn}} \sum_{f \in S(t)} \delta(t - t^{(f)}) \tag{3.20}$$

Here, $\delta(t)$ is Dirac's δ-function [*16]. Note that the derivative of $\Theta(t)$ is $\delta(t)$. If we discretize this with respect to time, we can solve it numerically as well as the membrane potential.

3.4.2 Conductance-based model and current-based model

As there is a difference between conductance-based and current-based models for neurons, there is a similar difference for synapses.

In all previous explanations, $g(t)$ was considered as a conductance value, and the synaptic current $I_{\mathrm{syn}}(t)$ had to be calculated based on equation (1.3). This is called a **conductance-based synapse**. It is often used in combination with conductance-based neurons (e.g. HH model).

On the other hand, $g_{\mathrm{syn}}(t)$ can be considered directly as the value of synaptic current, $I_{\mathrm{syn}}(t) = w \cdot g_{\mathrm{syn}}(t)$. In this case, the synaptic current depends on a constant w that represents the strength of the synaptic linkage, which is positive for excitatory connections and negative for inhibitory connections. This is called a **current-based synapse**. In the case of current-based synapses, the number of input spikes from the pre-side neuron is directly related to the strength of the current. In this case, we can use the same method as in the previous section. It is often used in combination with current-based neurons (e.g. LIF model).

This is not a matter of which one is better than the other, but it is better to use different methods depending on the situation. One criterion is whether or not the nature of synaptic currents due to reversal potential should be considered. In the case of inhibitory synapses In the case of inhibitory synapses, the current-based model allows the negative current to grow as large as possible in response to high frequency spiking inputs, while the conductance-based model prevents further current flow once the membrane potential is hyperpolarized to the reversal potential. In the next section, we will focus on the combination of the LIF model and current-based synapses.

3.4.3 Synaptic delay

When a pre-synaptic neuron fires a spike, the spike travels down the axon, and the conductance of the post-synaptic neuron changes only when a chemical is secreted from the synaptic terminal and

[*16] A function satisfying $\int_{-\infty}^{\infty} f(x)\delta(x)dx = f(0)$ for any continuous real function $f(x)$.

binds to a receptor. There is a time delay of a few milliseconds between these events. It is possible to incorporate such a **synaptic delay** explicitly. In this paper, we will basically focus on networks without delays, but simulations with delays will be explained later (Section 3.5.3).

3.5 Network simulation

Now that we know how to calculate synapses, let's create a network. The simplest network is one that consists of two neurons, so let's start there.

3.5.1 Simulation of two neurons

If the neurons are not connected by synapses and are completely independent, we can increase the number of neurons based on `lif.c` and increase the number of neurons. The code is `lif2.c`. Only the differences from li f . The only difference between `lif.c` and `lif2.c` is shown in Listing 3.5 below.

Listing 3.5 `diff -u lif.c lif2.c`

```
 1 --- lif.c       2021-04-26 22:39:04.000000000 +0900
 2 +++ lif2.c      2021-04-26 22:39:00.000000000 +0900
 3 @@ -13,21 +13,26 @@
 4  #define T       ( 1000.0 ) // ms; unused
 5  #define NT      ( 1000   ) // ( T / DT )
 6  #define I_EXT   (   12.0 ) // nA
 7 +#define N       (    2   ) // # of neurons
 8 
 9  int main ( void )
10  {
11 -  float v = V_REST;
12 +  float v [ N ] = { V_REST, V_REST - 15. };
13 +  bool s [ N ] = { false, false };
14 
15    for ( int32_t nt = 0; nt < NT; nt++ ) {
16      float t = DT * nt;
17 -    printf ( "%f␣%f\n", t, v );
18 +    printf ( "%f␣%f␣%f\n", t, v [ 0 ], v [ 1 ] );
19 
20 -    v += DT * ( - ( v - V_REST ) + R_M * I_EXT ) / TAU;
21 -    bool s = ( v > THETA );
22 +    for ( int32_t i = 0; i < N; i++ ) {
23 +      v [ i ] += DT * ( - ( v [ i ] - V_REST ) + R_M * I_EXT ) / TAU;
24 +      s [ i ] = ( v [ i ] > THETA );
25 +    }
26 
27      // Pretty-print spikes on membrane potentials. Note that spike time is not t but t + DT
28 -    if ( s ) { printf ( "%f␣%f\n%f␣%f\n", t + DT, v, t + DT, 0. ); }
29 +    if ( s [ 0 ] ) { printf ( "%f␣%f␣%f\n%f␣%f␣%f\n", t + DT, v [ 0 ], v [ 1 ], t + DT, 0., v [
       1 ] ); }
30 +    if ( s [ 1 ] ) { printf ( "%f␣%f␣%f\n%f␣%f␣%f\n", t + DT, v [ 0 ], v [ 1 ], t + DT, v [ 0 ],
        0. ); }
31 
32 -    v = s * V_RESET + ( ! s ) * v;
33 +    for ( int32_t i = 0; i < N; i++ ) { v [ i ] = s [ i ] * V_RESET + ( ! s [ i ] ) * v [ i ]; }
34    }
35  }
```

In line 7, we define the number of neurons N as $\texttt{N} = 2$. In line 12, we initialize the membrane potential, where the variable v is an array. We initialize one of the membrane potentials to V_{rest} (mV) and the other to $V_{\mathrm{rest}} - 15$ (mV). In line 13, the variable `s`, which stores the spikes, is also defined as an array. We also change the display of the membrane potential in line 18. The membrane potential estimation and spike generation are also repeated in `for` loops since there are two neurons (lines 22–25). The spike waveform (lines 29 and 30) is displayed for each neuron. The update of membrane potential (line 33) is also repeated in `for` loop.

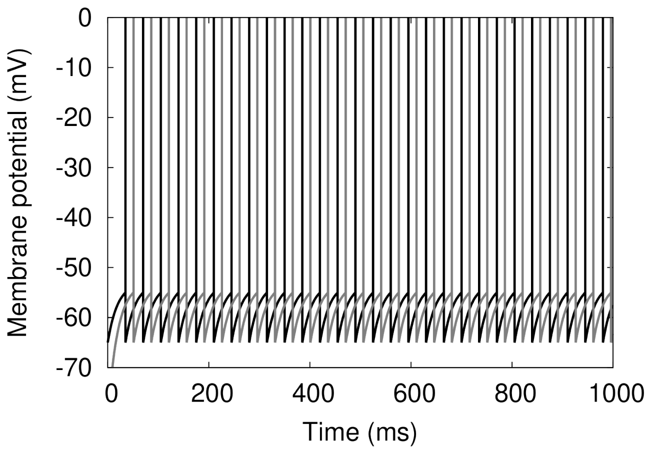


Fig. 3.11 Plot of the membrane potential of two neurons. The view of the figure is the same as in Figure 3.7. Black and gray represent different neurons.

When we run the simulation, we obtain a plot of membrane potentials as in Figure 3.11. The two neurons have different initial membrane potential values, so the timing of the spike firing is different, but other than that, the behavior is the same. to display two separate columns simultaneously in `gnuplot`, use `using` as follows.

```
node00:~/snsbook/code/part1/lif$ make lif2
:
node00:~/snsbook/code/part1/lif$ ./lif2 > lif2.dat
node00:~/snsbook/code/part1/lif$ gnuplot
:
gnuplot> plot 'lif2.dat' using 1:2 with lines, 'lif2.dat' using 1:3 with lines
```

3.5.2 Network of two neurons

Let's build a minimal network by combining these two neurons with the simplest current-based exponential decay synapses. The code is `network.c`. Only the differences from `lif2.c` are shown in Listing 3.6 below.

Listing 3.6 **`diff -u lif2.c network.c`**

```
--- lif2.c      2021-04-26 22:39:00.000000000 +0900
+++ network.c   2021-04-26 22:38:33.000000000 +0900
@@ -14,18 +14,23 @@
 #define NT      ( 1000   ) // ( T / DT )
 #define I_EXT   (   12.0 ) // nA
 #define N       (    2   ) // # of neurons
+#define TAU_SYN (    5.0 ) // ms
+#define R_SYN   (    1.0 ) // MOhm
+#define W       (    2.0 ) // connection weight

 int main ( void )
 {
   float v [ N ] = { V_REST, V_REST - 15. };
+  float i_syn [ N ] = { 0., 0. };
   bool s [ N ] = { false, false };
```

```
16
17     for ( int32_t nt = 0; nt < NT; nt++ ) {
18       float t = DT * nt;
19       printf ( "%f␣%f␣%f\n", t, v [ 0 ], v [ 1 ] );
20
21 +     for ( int32_t i = 0; i < N; i++ ) { i_syn [ i ] = exp ( - DT / TAU_SYN ) * i_syn [ i ] + W *
          s [ ( i + 1 ) % 2 ]; }
22       for ( int32_t i = 0; i < N; i++ ) {
23 -       v [ i ] += DT * ( - ( v [ i ] - V_REST ) + R_M * I_EXT ) / TAU;
24 +       v [ i ] += DT * ( - ( v [ i ] - V_REST ) + R_SYN * i_syn [ i ] + R_M * I_EXT ) / TAU;
25         s [ i ] = ( v [ i ] > THETA );
26       }
```

The changes to the code are as follows: Lines 7–9 define the synaptic constants. The time constant is 5 ms (`TAU_SYN`), the formal resistance of the synaptic current is 1 MΩ (`R_SYN`), no reversal potential is introduced since we are using current-based synapses, and the coupling strength `W` is 2.0. This is a positive value, so it is an excitatory synaptic coupling. The synaptic currents are calculated on line 21, and are added to the estimate of the membrane potential on line 24.

The calculation of the synaptic currents (line 21) is the central part of the process, which can be done using the method introduced in Section 3.4.1 to calculate the opponent's spike `s [ ( i + 1 ) % 2 ]` Calculate the current using [ ]. Note that if you are neuron i, your opponent is neuron $(i+1)\%2$ (% 2 is the remainder divided by 2). Multiply this by the bond strength `W`.

When we simulate this network, we can see how the timing of the spikes gradually align (Figure 3.12A). We call such a state **synchronous firing** or **synchronous state**. On the other hand, when the sign of the coupling strength W is negative and the inhibitory synapses are coupled to each other, the timing of the spikes gradually becomes alternating (Figure 3.12B). This state is sometimes referred to as a **reverse-phase synchronization state**. In this way, the behavior of a network of even two simple neurons can change drastically just by changing the way they are coupled. Since this is the case even with two neurons, it is not difficult to imagine the enormous number of patterns that can be generated in a brain as large as the human brain, which has 86 billion neurons.

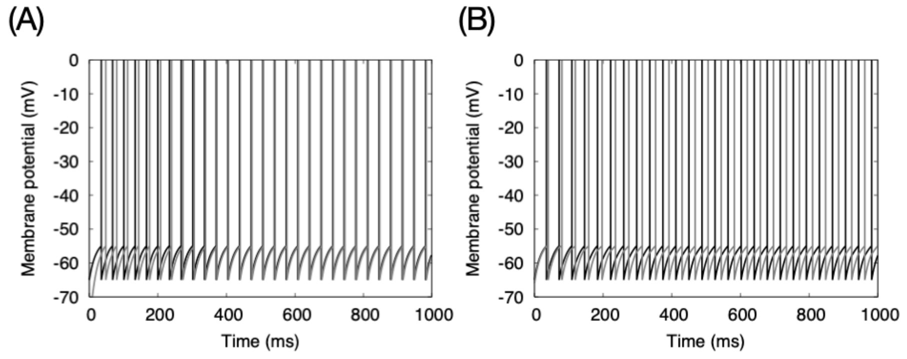


Fig. 3.12 Simulation of a network of two neurons. (A) Excitatory connections between neurons. (B) When the neurons are inhibitively coupled to each other. The initial value of the membrane potential is shifted by a large value (−15 mV) in (A) and a small value (−1 mV) in (B) to make it easier to see the changes. The view is the same as in Figure 3.7.

3.5.3 Simulation incorporating synaptic delay *

It is also possible to incorporate synaptic delay explicitly. For simplicity, we will assume that a neuron fires at most one spike during a synaptic delay. This assumption is only valid if the refractory period is longer than the synaptic delay, and by making this assumption, we do not need to keep information about earlier spikes *[17]. 5 ms refractory period and 2 ms synaptic delay are both incorporated in the code `network_delay.c`. Only the differences between `network.c` are shown in Listing 3.7 below.

Listing 3.7 `diff -u network.c network_delay.c`

```
 1 --- network.c   2021-04-26 22:38:33.000000000 +0900
 2 +++ network_delay.c     2021-04-26 22:38:26.000000000 +0900
 3 @@ -17,27 +17,37 @@
 4  #define TAU_SYN (    5.0 ) // ms
 5  #define R_SYN   (    1.0 ) // MOhm
 6  #define W       (    2.0 ) // connection weight
 7 +#define T_REFR  (    5.0 ) // ms; unused
 8 +#define NT_REFR (    5   ) // ( T_REFR / DT )
 9 +#define DELAY_SYN  ( 2.0 ) // ms; unused
10 +#define NDELAY_SYN ( 2   ) // ( DELAY_SYN / DT )
11
12  int main ( void )
13  {
14    float v [ N ] = { V_REST, V_REST - 15. };
15    float i_syn [ N ] = { 0., 0. };
16    bool s [ N ] = { false, false };
17 +  int32_t ts [ N ] = { 0, 0 };
18 +  int32_t refr [ N ] = { 0, 0 };
19
20    for ( int32_t nt = 0; nt < NT; nt++ ) {
21      float t = DT * nt;
22      printf ( "%f␣%f␣%f\n", t, v [ 0 ], v [ 1 ] );
23
24 -    for ( int32_t i = 0; i < N; i++ ) { i_syn [ i ] = exp ( - DT / TAU_SYN ) * i_syn [ i ] + W *
        s [ ( i + 1 ) % 2 ]; }
25 +    for ( int32_t i = 0; i < N; i++ ) { i_syn [ i ] = exp ( - DT / TAU_SYN ) * i_syn [ i ] + W *
        ( ts [ ( i + 1 ) % 2 ] + NDELAY_SYN == nt ); }
26      for ( int32_t i = 0; i < N; i++ ) {
27        v [ i ] += DT * ( - ( v [ i ] - V_REST ) + R_SYN * i_syn [ i ] + R_M * I_EXT ) / TAU;
28        s [ i ] = ( v [ i ] > THETA );
29 +      ts [ i ] = ( s [ i ] ) * ( nt + 1 ) + ( ! s [ i ] ) * ts [ i ];
30      }
31
32      // Pretty-print spikes on membrane potentials. Note that spike time is not t but t + DT
33      if ( s [ 0 ] ) { printf ( "%f␣%f␣%f\n%f␣%f␣%f\n", t + DT, v [ 0 ], v [ 1 ], t + DT, 0., v [
       1 ] ); }
34      if ( s [ 1 ] ) { printf ( "%f␣%f␣%f\n%f␣%f␣%f\n", t + DT, v [ 0 ], v [ 1 ], t + DT, v [ 0 ],
        0. ); }
35
36 -    for ( int32_t i = 0; i < N; i++ ) { v [ i ] = s [ i ] * V_RESET + ( ! s [ i ] ) * v [ i ]; }
37 +    for ( int32_t i = 0; i < N; i++ ) {
38 +      refr [ i ] = s [ i ] * ( NT_REFR ) + ( ! s [ i ] ) * ( refr [ i ] - 1 );
39 +      v [ i ] = ( refr [ i ] > 0 ) ? V_RESET : v [ i ];
40 +    }
41    }
42  }
```

Lines 7 and 8 define the length of the refractory period, and lines 9 and 10 define the synaptic latency. lines 17 and 18 provide a variable `ts` to hold the firing time of the previous spike, and a counter `refr` for the refractory period. line 25 calculates the synaptic current, adding the spike if "spike time of the pre-side neuron + synaptic latency = current time". Line 29 stores the last spike time. Lines 37–40 update the refractory period and membrane potential.

*[17] Because once the synaptic delay time has passed since the spike was fired, you can forget about that spike.

3.5.4 Random Network Simulation

The network of two neurons is so small that even an ordinary laptop can finish the calculation in a matter of seconds. This is no fun. Since this is not interesting, let's consider a larger network. Specifically, we consider a random network of 4000 neurons that are randomly combined with a 4:1 excitatory:inhibitory distribution with probability $p = 0.02$ (Brian Simulator, 2017). This network was originally developed to study the neural activity caused by the interaction of excitatory and inhibitory neurons in the cerebral cortex, but is now also used as a benchmark to test the computational performance of various neural circuit simulators on supercomputers (Brette et al., 2007).

The equations for the membrane potential are the same as in equations (3.9)–(3.11).

$$\begin{aligned} \tau \frac{dv}{dt} &= -\left(v(t) - V_{\text{rest}}\right) + ge(t) + gi(t) \\ v(t) &> \theta \Rightarrow S(t) = 1, v(t) \leftarrow V_{\text{reset}} \\ v(0) &= V_{\text{init}} \end{aligned}$$

where $v(t)$ is the membrane potential at time t, $\tau = 20$ ms is the time constant, $V_{\text{rest}} = -49$ mV is the resting potential, $ge(t)$ and $gi(t)$ are the excitatory and inhibitory synaptic currents, respectively, $\theta = -50$ mV is the threshold for spike firing, $V_{\text{reset}} = -60$ mV is the reset potential, $V_{\text{init}} = -60 + 10 \times \text{rand}(t)$ is the initial value of the membrane potential, and rand(t) is a uniform random number in $[0, 1)$. On the other hand, the synaptic current is calculated by the following equation:

$$\begin{aligned} ge(t) &= \exp\left(-\frac{\Delta t}{\tau_e}\right) ge(t - \Delta t) + w_e \sum_{j \in \text{Exc}} \text{S}_j(t), \\ gi(t) &= \exp\left(-\frac{\Delta t}{\tau_i}\right) gi(t - \Delta t) + w_i \sum_{j \in \text{Inh}} \text{S}_j(t) \end{aligned} \tag{3.21}$$

where $\tau_e, \tau_i = 5and10$ ms are time constants, respectively, Exc, Inh are excitatory and inhibitory neuron populations, respectively, $w_e = 1.62/\tau_e$, $w_i = -9/\tau_i$ mV is the change in postsynaptic potential per spike input, respectively, and $\text{S}_j(t) \in \{0, 1\}$ is 1 if neuron j fires a spike at time t and 0 otherwise.

The code is `code/part1/random/random.c`. If you compile and run the code as follows, you will get a file named `spike.dat`, and if you display it with `gnuplot`, you will get a spike time plot (**raster plot**) as shown in Figure 3.13.

```
node00:~/snsbook/code/part1/random$ make
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -DSFMT_MEXP=19937 -c random.c
:
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -DSFMT_MEXP=19937 -o random random.o SFMT.o
     timer.o -lm
node00:~/snsbook/code/part1/random$ ./random
Elapsed time = 16.529303 sec.
node00:~/snsbook/code/part1/random$ gnuplot
:
gnuplot> plot 'spike.dat' with dots
```

Plotting the membrane potential of 4000 neurons at once doesn't look right, so from now on we will only plot the spike times like this Plotting [*18].

[*18] See Appendix A.2 for plotting instructions.

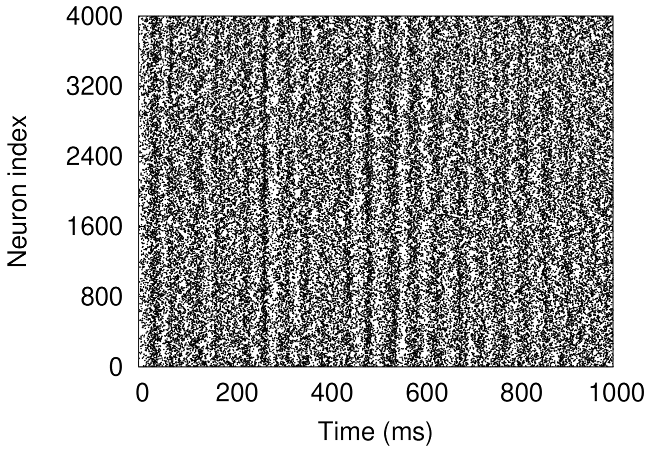


Fig. 3.13 Raster plot of a random network. The dots represent the time of spike firing.

A summary of the code is as follows:

Listing 3.8 `random.c:loop`

```
109  for ( int32_t nt = 0; nt < NT; nt++ ) {
110    calculateSynapticInputs ( n );
111    updateCellParameters ( n );
112    outputSpike ( nt, n );
113  }
```

In the time loop (line 109), we calculate the synaptic input (line 110) and then update the value of the membrane potential (line 111). After the neurons have been calculated, output the spike information to a file (line 112).

Using the authors' computer (Intel Xeon E5-2650 v4 2.2GHz) as usual, it took 17 seconds for one simulation. This is a reasonable amount of computation time for a network of this size, but if the network is made larger, the computation time will be enormous. If we want to study neural activity during learning, we need to run simulations for several minutes to several hours instead of one second, the computation time will increase proportionally. One method to speed up the computation is **parallel computing**, which will be introduced in Part III.

3.6 Analysis method for spike trains

The raster plot is good for visual overview of neuron activity, but for quantitative discussion, we need to perform quantitative/statistical analysis of the spike trains. We have already calculated the firing frequency of a single neuron to draw the I-F curve (Section 3.1.3), but since we have just obtained the firing pattern of a population of neurons, let's take it a step further. For spike statistics, Rieke et al. (1997) is a good book that explains the basics well.

3.6.1 Firing rate

There are many ways to calculate the firing frequency f. The firing frequency calculated in section 3.1.3 is precisely called the **time-averaged firing rate**, where $f = N/T$ for N spikes fired in T seconds [*19]. In the case of the neuron in Figure 3.12, there were 29 spikes fired in 1000 ms, which

[*19] $T < 1$ is also acceptable.

is 29 spikes/s [*20].

The time-averaged firing frequency is assumed to be averaged over a sufficiently long time interval. This is ideal if the neuron fires spikes at a constant rate throughout the stimulus, but if the neuron fires spikes at a very high rate for a very short period of time and then immediately stops (called **burst firing**), information about the temporal variation of firing frequency is lost.

Suppose that two consecutive spikes $f_1 and f_2$ are fired at time t_1 and t_2. The time interval between the 2 spikes (i.e., $t_2 - t_1$) is called the **inter-spike interval (ISI)**, and its inverse (i.e., $1/(t_2 - t_1)$) is called an **instantaneous firing rate**. If a neuron fires at a fixed period, the time-averaged firing frequency and the instantaneous firing frequency coincide. For example, in the example, the ISI was 34 ms, so $1000/34 \approx 29$ spikes/s.

3.6.2 Cross-correlation

In the example in Figure 3.12, the two neurons both fired at 29 spikes/s. However, by changing the way the network was connected, the neurons fired at the same time and vice versa. However, by changing the way the network was connected, the neurons fired at the same time or at the opposite time. This difference in network behavior cannot be expressed by the measure of firing frequency.

The concept of **correlation** is used to show the relationship between two neurons. Consider the time interval $[0, T]$, and suppose that the spike trains of neuron $i \in \{1, 2\}$ are given as $S^{(1)}(t)$, $S^{(2)}(t)$, where $S^{(i)}(t)$ is 1 if neuron i fires at time t and 0 is otherwise. We discretize the spike sequence in terms of time, Δt and define the number of intervals as the $N = T/\Delta t$. Then, we define a **cross-correlation** $C^{(1,2)}(\tau)$ as follows:

$$C^{(1,2)}(\tau) = \sum_{n=1}^{N} S^{(1)}(n) S^{(2)}\left(n + \frac{\tau}{\Delta} t\right) \tag{3.22}$$

where τ is the time difference. The cross-correlation indicates how long it takes for the firings of the two neurons to align (or not).

The cross-correlation of Figure 3.12 is shown in Figure 3.14. The seemingly synchronized case (Fig. 3.12A, in-phase) actually fires after a delay of $\tau = 4$ ms, and the reverse timing case (Fig. 3.12B, anti-phase) fires after a delay of $\tau = 17$ ms.

The correlation with itself is called **autocorrelation**, and it has a maximum value at $\tau = 0$. When firing occurs periodically, the peak of the autocorrelation appears with a time delay corresponding to the period of the firing, so it is used to determine whether the firing is periodic or not.

3.6.3 Population average and trial average

Next, let's consider a random network of 4000 neurons (Figure 3.13). In this case, we can also consider the time averages for each individual neuron, but this time we can calculate the **population average** firing frequency for a group of neurons (Figure 3.15). The spike trains of neurons from the beginning of the stimulus are lined up, and an appropriate time width (bin width) Δ is considered, and the time interval is divided by Δ into n bins. The number of spikes in each bin is counted and divided by Δ and the number of neurons. Count the number of spikes in each bin and divide by Δ and the number of neurons to calculate the number of spikes per neuron per unit time, or the population average firing frequency.

For the raster plot in Figure 3.13, we calculated the population mean (Figure 3.16). The plots show what happens when the bin width is changed to 1 ms, 10 ms, and 100 ms. When the bin width

[*20] When calculating this, the simulation was run for 2000 ms, and the data for the first 1000 ms was discarded and only the data for the second 1000 ms was used. Since the beginning of the simulation contains transient responses that depend on the initial values, it is necessary to wait until the behavior of the network becomes steady when doing this kind of analysis.

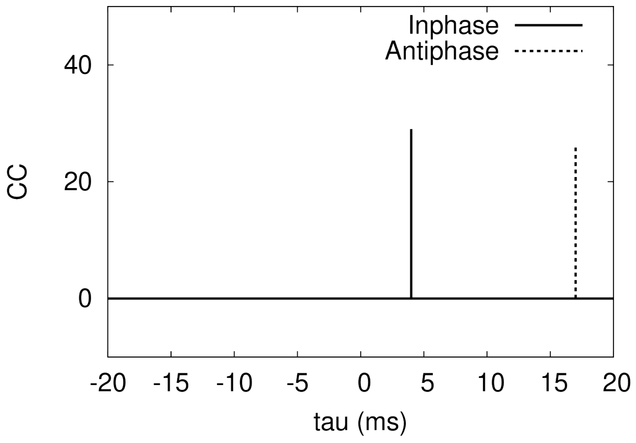


Fig. 3.14 Cross-correlation in the two neurons in Figure 3.12

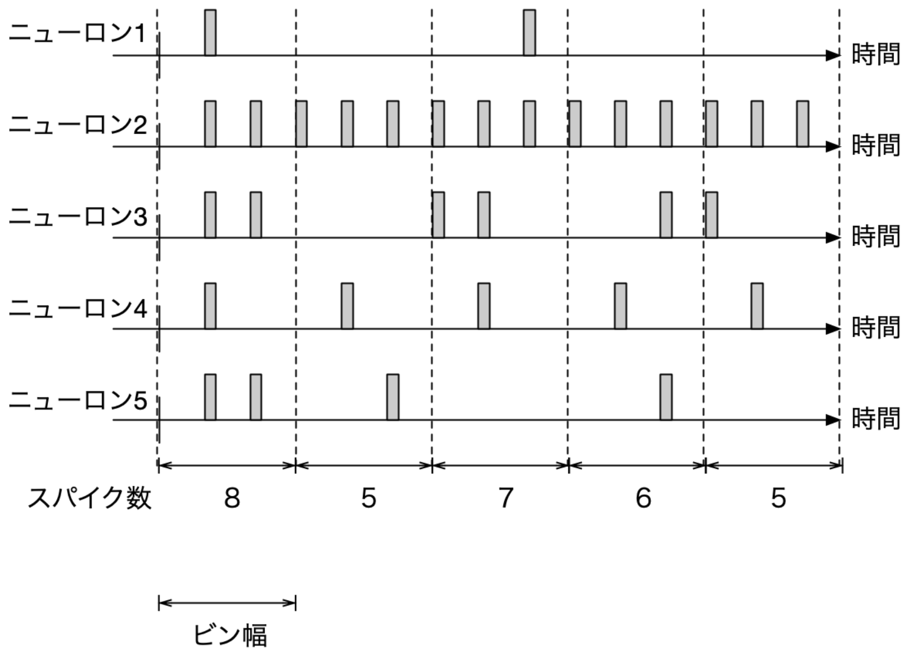


Fig. 3.15 Calculation of the population average firing frequency. A spike train of five neurons is used as an example. The gray vertical bars represent firings.

is too small, the instantaneous increase in firing frequency appears randomly. When the bin width is too large, the firing frequency is almost constant in time. At intermediate bin widths, peaks appear every 50 ms, indicating that the firings of the neuron population are periodically synchronized. Thus, when calculating the population average, it is necessary to adjust the bin width according to the time scale to be studied.

In the above, we considered the (population average) firing frequency of multiple neurons in a single trial. On the other hand, there are cases where we want to know the average firing frequency between trials by repeatedly measuring the response of a single neuron to a stimulus in multiple trials. This is because the behavior of a neuron differs slightly from trial to trial even when the same stimulus is given due to fluctuations in the membrane potential of the neuron or the state of the ion channel. This type of averaging is called **trial average**. The method of calculation is almost the

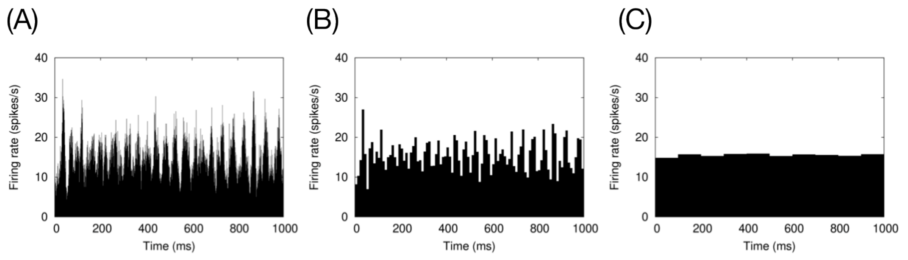


Fig. 3.16 Population means of excitatory neurons in a random network. (A–C) Plotted for different bin widths Δ (A: 1 ms; B: 10 ms; C: 100 ms).

same as in the case of population averaging, and in the example of Fig. 3.16, the mean of neurons 1–5 is The spike sequence can be replaced by neuron 1 (for example) in trials 1–5, where the stimulus is given at time 0. The resulting histogram has the name **PSTH** (Peri-Stimulus Time Histogram); an example of PSTH calculation is given in Chapter 6. In PSTH, it is also important to choose just the right bin width. A method has been proposed to find the optimal bin width by assuming some kind of stochastic process in the firing of neurons (Shimazaki and Shinomoto, 2007).

3.6.4 Coefficient of Variation

In section 3.3.2, we introduced a method for generating Poisson spikes. The Poisson spike sets only the average firing frequency, and the actual timing of the spikes is random. How random is the spike sequence? As a measure for calculating how random a spike sequence is, a value called the **coefficient of variation (CV)** has been defined.

Given a sequence of N spikes, $S = \{f_1, \cdots, f_N\}$, and their firing times are $t_1, \ldots, t_N$ where N is a sufficiently large value. For two consecutive spikes f_i, f_{i+1}, define their interval as $\mathrm{ISI}_i = t_{i+1} - t_i$. If the mean and standard deviation of $\mathrm{ISI}_1, \ldots, \mathrm{ISI}_{N-1}$ are μ and σ, respectively, then CV is calculated as

$$\mathrm{CV} = \frac{\sigma}{\mu}. \tag{3.23}$$

In a stochastic process following the Poisson process, it has been proven that $\mu = \sigma$, so the CV of a Poisson spike is 1. On the other hand, for a spike train fired at regular intervals, $\sigma = 0$, so the CV is 0. The value of CV varies from 0 to 1, and larger value indicates more random for spike trains.

3.7 Simulation that takes into account even the shape of the neuron *

Finally, let's discuss a simulation that takes into account the spatial geometry of neurons.

In the previous explanations, we have ignored the spatial shape of the neuron and simply considered it as a point in space. Even if a neuron is represented as a point, its basic properties, such as receiving multiple inputs, summing them, determining whether a threshold is exceeded by the membrane potential, and spiking can be implemented. In other words, the behavior of a network is more important than the information processing of individual neurons. On this basis, a neuron has been described as having a single membrane potential and a set of ion channels and synaptic connections that depend on that membrane potential.

On the other hand, an actual neuron has a characteristic spatial shape (Fig. 3.17), and the type of ion channel on the membrane surface and the value of the membrane potential vary depending on the position. By taking these shapes into account, it has been suggested that a single neuron may be able to perform much more sophisticated information processing than previously thought (Segev et al., 1994; Koch and Segev, 1998). For example, when multiple locations are stimulated in sequence,

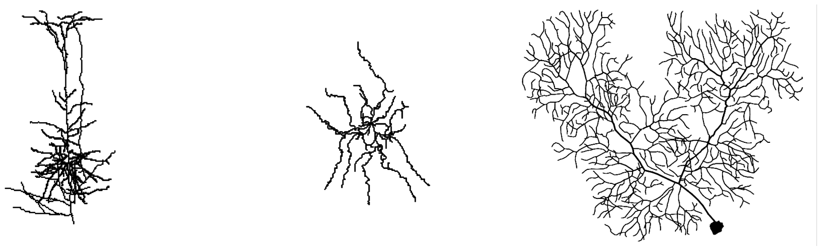


Fig. 3.17 Spatial geometry of various neurons. Left: cortical pyramidal cells (Trevelyan et al., 2006). Center: medium-sized spinous cells in the basal ganglia (Martone et al., 2003). Right: cerebellar Purkinje cells(Anwar et al., 2014). Image obtained from NeuroMorpho.Org (Ascoli, 2006) (NMO_02236, NMO_04519, NMO_35058).

the way the neurons respond changes depending on the order of the stimuli (Rall, 1964; Branco et al., 2010). This property makes it possible to discriminate between stimulus sequences, but not between stimuli if they do not have a shape. Recently, for example, Gidon et al. (2020) showed both experimentally and in simulation that nonlinear computation on dendrites can be used to perform a logical operation called XOR (exclusive OR), which is an operation that can be learned by multilayer perceptrons (Moldwin and Segev, 2020). This is because XOR is an operation that can be learned by multilayer perceptrons (Moldwin and Segev, 2020). If this is the case, the brain can be thought of as a deep network of many neural nets, and its computational power can be imagined to be very powerful [*21].

3.7.1 Modeling methods

In the case of spatially shaped neurons, currents flow through the cell, so each location in the cell has a different value of membrane potential. In addition, each location can have independent ion channels and synaptic connections. In this case, the value of the membrane potential is written as $V(x,t)$, a function of intracellular position and time. The value of the membrane potential is written as a function of intracellular position and time, $V(x,t)$, where x is the spatial position in the cell and t is the time. The membrane potential $V(x,t)$ is calculated by the equation (3.24).

$$C_m \frac{\partial V}{\partial t} = \frac{\lambda^2}{R_m}\frac{\partial^2 V}{\partial x^2} - I_{\text{ion}}(x,t) + I_{\text{syn}}(x,t) + I_{\text{ext}}(x,t) \tag{3.24}$$

where C_m is the capacitance per unit area, λ is the **space constant** (see below), R_m is the membrane resistance per unit area, $I_{\text{ion}}(x,t)$ is the ionic current of the membrane, $I_{\text{syn}}(x,t)$ is the synaptic current, and $I_{\text{ext}}(x,t)$ is the external current. Note that with respect to current, it is also a function of position x. Since the membrane potential is a function of space and time, the derivative is a partial derivative. Except for the first term on the right-hand side, equation (3.24) can be expressed as

It is not so different from the equation for the membrane potential in The only difference is that the variable x is increased. The important thing is that the first term on the right side $\frac{\lambda^2}{R_m}\frac{\partial^2 V}{\partial x^2}$. The term $\frac{\partial^2 V}{\partial x^2}$ is called the **diffusion term**, and it represents the effect that if the membrane potential is different at two different points in space, a current is generated depending on the potential difference. The distance constant The space constant, λ, means that the membrane potential decays to $1/e$ as

[*21]We don't know what we can actually do and how much we can do.

the distance, λ, increases [*22].

It is important to note here that the parameters of the cells have spatial units. For example, the HH model also had a unit of $1/\mathrm{cm}^2$ attached to the parameters, but for calculation purposes, it was enough that the units of the left and right sides were the same. Now that we are considering the shape of the cell, we need to take into account the surface area of the membrane and the length of the dendrites.

This section explains how to process the key point $\frac{\partial^2 V}{\partial x^2}$. Basically, it's the same as time differentiation, just take the spatial difference. In order to simplify the process, first consider a single cable with no branches, and then think about how the current flows on it. Let's divide into N parts in the small interval Δx, and write the membrane potential at each point as $V(x,t)$.

$$\frac{\partial^2 V}{\partial x^2} \approx \frac{\partial}{\partial x}\frac{V(x+\Delta x/2,t)-V(x-\Delta x/2,t)}{\Delta x}$$

and differencing. If we pay attention to the way the difference is taken here, it is calculated by shifting $\Delta x/2$ to the left and right with position x at the center. This method is called **central difference**. Repeating the central difference again, we obtain the following:

$$\begin{aligned}\frac{\partial^2 V}{\partial x^2} &\approx \frac{\partial}{\partial x}\frac{V(x+\Delta x/2,t)-V(x-\Delta x/2,t)}{\Delta x}\\ &= \frac{1}{\Delta x}\left(\frac{\partial V(x+\Delta x/2,t)}{\partial x}-\frac{\partial V(x-\Delta x/2,t)}{\partial x}\right)\\ &\approx \frac{1}{\Delta x}\left(\frac{V(x+\Delta x,t)-V(x,t)}{\Delta x}-\frac{V(x,t)-V(x-\Delta x,t)}{\Delta x}\right)\\ &= \frac{V(x+\Delta x,t)-2V(x,t)+V(x-\Delta x,t)}{\Delta x^2}\end{aligned}$$

In other words, from a point x, current flows according to the potential difference between the two points $x+\Delta x$ and $x-\Delta x$.

This is a long explanation, but based on this idea, we can consider the following modeling (Figure 3.18). First, divide the neuron (Fig. 3.18A) into short cables that are electrically uniform [*23] and connect them with resistors (Fig. 3.18B). The length of the cable is l, the diameter is d, and the resistance in the longitudinal direction in the cable is R_a. The value of λ is calculated as $\lambda = \sqrt{dR_m/2R_a}$. This cable is called a **compartment**. If the index of a compartment is μ, the membrane potential $V_i(t)$ of compartment i in the absence of bifurcation (Fig. 3.18C) follows the equation:

$$C_m\frac{\partial V_\mu}{\partial t} = g_{\mu,\mu+1}(V_{\mu+1}-V_\mu) - g_{\mu,\mu-1}(V_\mu - V_{\mu-1}) - I_{\mathrm{ion}}(\mu,t) + I_{\mathrm{syn}}(\mu,t) + I_{\mathrm{ext}}(\mu,t) \tag{3.25}$$

where $g_{i,j}$ is the conductance between compartments i and j. The resistance between these two compartments is

$$\frac{R_a}{2}\frac{l_i}{\pi d_i^2} + \frac{R_a}{2}\frac{l_j}{\pi d_j^2} \tag{3.26}$$

Therefore, by taking the reciprocal of this value and dividing by the surface area of compartment i, $\pi d_i l_i$, the conductance value between the two compartments, $g_{i,j}$, can be calculated. In other words

$$g_{i,j} = \frac{d_i d_j^2}{R_a l_i \left(l_i d_j^2 + l_j d_i^2\right)} \tag{3.27}$$

[*22] It corresponds to the fact that the time constant represents the decay of the membrane potential with respect to time.

[*23] Meaning that there is no potential gradient inside it.

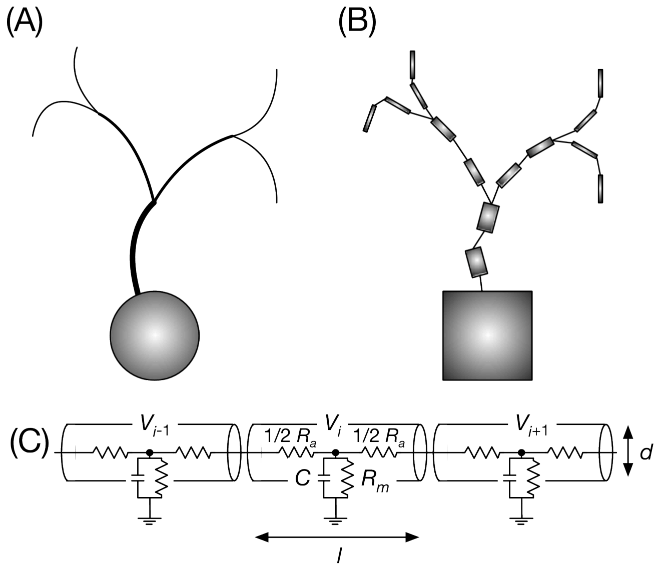


Fig. 3.18 Schematic of the multicompartment model. (A) Shape of the target neuron. (B) Representation of compartments by their connections. (C) Electrical coupling between compartments.

We can write If there is a branch, the first two terms on the right-hand side of equation (3.25) increase to three terms.

This type of models is called **multi-compartment model**, because it consists of multiple electrically homogeneous compartments. Each compartment has its own membrane potential, ion channels, and synaptic connections, and thus corresponds to a single neuron without considering its shape [*24] . This means that the multicompartment model can be viewed as an electrically coupled single neuron. In other words, if a neuron with a single shape is composed of 100 compartments, the computation is almost equivalent to simulating 100 neurons with no shape and calculating the interaction between them via conductance [*25].

3.7.2 Simulation of a multi-compartment model

As an example of a multi-compartment model, let us try to implement the hippocampal CA3 model [*26] of Traub and Wong (1991). This model consists of 19 one-dimensional compartments, each of which has an independent size and length, and each of which is assigned a membrane potential and an ion channel (Figure 3.19). The details of the ion channel equations and various parameters are enormous, so please refer to the original paper [*27].

The code was prepared in `code/part1/multi/traub.c` [*28]. Compiling and running the code as follows will run the simulation when a current of 0.1 nA is input to the cell body. Plotting the membrane potential of the cell body (compartment 8) gives a characteristic spike waveform (Figure 3.20A).

```
node00:~/snsbook/code/part1/traub$ make
```

[*24]A neuron model that does not take into account shape is a **single-compartment model**.

[*25]In practice, calculations are heavier in the multi-compartmental case because the numerical methods used are often changed.

[*26]One of the areas in the hippocampus that receives input. See Section 8.1.

[*27]PDF is available at `https://www.researchgate.net/publication/21491281`.

[*28]The model is simple enough to be solved by the Euler method.

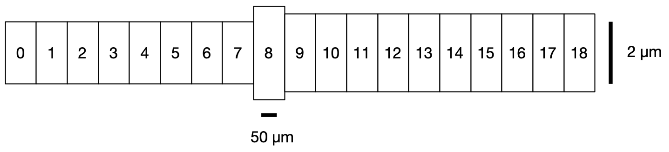


Fig. 3.19 Spatial geometry of neurons in the hippocampal CA3 model. The rectangles represent each compartment, and the numbers represent the compartment numbers.

```
gcc -O3 -std=gnu11 -Wall -c traub.c
gcc -O3 -std=gnu11 -Wall -o traub traub.o -lm
node00:~/snsbook/code/part1/traub$ ./traub > traub.dat
node00:~/snsbook/code/part1/traub$ gnuplot
:
gnuplot> plot 'traub.dat' using 1:10 with lines
```

Note that the first column of `traub.dat` is the time, and the value of the membrane potential in compartment 8 is in the tenth column. When a current of 10 nA is applied for 1 ms at time 1000 ms, a spike is instantly fired in the cell body, and it propagates backwards while decaying on the dendrites (Figure 3.20B). The multicompartment model allows us to study the spatiotemporal behavior of the membrane potential of a neuron with such a spatial extent.

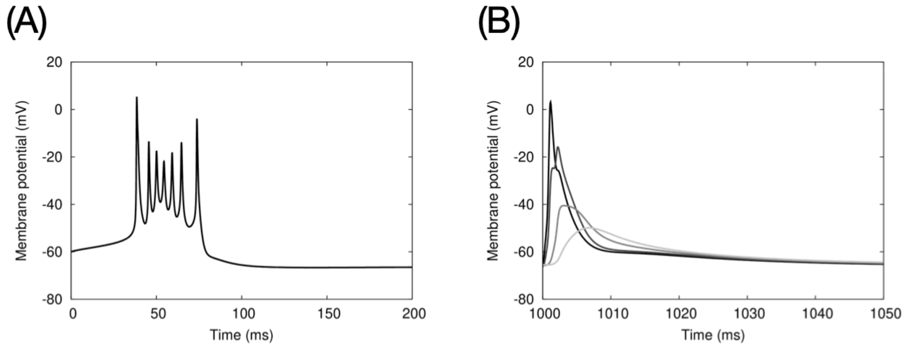


Fig. 3.20 Waveform of the membrane potential of the hippocampal CA3 model (Traub and Wong, 1991). (A) Membrane potential in the cell body when a constant current is applied to the cell body. (B) Membrane potential in the cell body and on dendrites when a pulsed current is applied to the cell body. Starting from the cell body, the membrane potential is shown for each of the three compartments (black to gray).

3.8 Column: How to write `Makefile`

If the size of your code is small and it can fit in a single file, you can get by with manually typing `gcc -O3 -o ...`. However, as the scale of the program increases, you may want to split the file. Or, even if the scale is not so large, for example, you may want to define a separate file for each cell type. It is good to be able to use split compilation because it increases the readability of the program and the productivity of the development.

The first technology that supports split compilation is `make`. By using this technology, you just need to type `make`, and the rest are done automatically. The `Makefile` describes the behavior of `make`.

First, let's prepare a template for split compilation. I prepared the code under `code/column/make/`.

Listing 3.9 `main.c`

```
#include <stdio.h>
#include "skel.h"

int main ( void )
{
  foo ( );
}
```

The `main` function calls the function `foo`. `foo` is defined in `skel.c` and `skel.h`.

Listing 3.10 `skel.c`

```
#include <stdio.h>
#include "skel.h"

void foo ( void ) { /* ... */ }
```

I'm just preparing the template now, so `foo` won't do anything for now.

Listing 3.11 `skel.h`

```
#pragma once
extern void foo ( void );
```

The second line is the `extern` declaration of the function, and `main.c` knows the function exists by looking at this declaration. The first line is slightly explained as follows: In the past, it was conventionally written as follows:

```
#ifndef __SKEL_H__
#define __SKEL_H__

extern void foo ( void );

#endif // __SKEL_H__
```

`skel.h` is referenced by both `main.c` and `skel.c` during compilation, but if the same definition or declaration is repeated many times, the compiler will stop with an error because it is a duplicate. This special way of writing was used to ensure that definitions and declarations were made only once during compilation. However, all major compilers now support `#pragma once`, and this one line will do the same thing.

The `Makefile` to compile them looks like this.

Listing 3.12 `Makefile.1`

```
CC = gcc
```

```
 2 CFLAGS = -O3 -std=gnu11 -Wall
 3
 4 all: main
 5
 6 main: main.o skel.o
 7         $(CC) $(CFLAGS) -o $@ $^
 8
 9 main.o: main.c skel.h
10         $(CC) $(CFLAGS) -c $<
11
12 skel.o: skel.c skel.h
13         $(CC) $(CFLAGS) -c $<
14
15 clean:
16         rm -f main *.o
```

Lines 1 and 2 specify the compiler and compile options, and line 4 specifies the

```
4 all: main
```

The left side of the colon (`all`) is called the **target** and the right side (`main`) is called the **dependency**. `make` without any arguments executes the first target in `Makefile`. Line 6 shows the dependency of mai n, and line 7 shows the command to be executed after the dependency is resolved. The meaning of lines 6 and 7 is that after `main.o` and `skel.o` have been created, the

```
7         $(CC) $(CFLAGS) -o $@ $^
```

Here, `$@` and `$^` are called **auto varibles**, which are expanded into target and dependency, respectively. Therefore, this command is called

```
        gcc -O3 -std=gnu11 -Wall -o main main.o skel.o
```

Similarly, lines 9 and 10 are instructions to generate `main.o`, and lines 12 and 13 are instructions to generate `skel.o`, respectively. Those include `skel.h`, so this is added to the depenency. The automatic variable `$<` is expanded to the first element of the dependency. Thus, for example, line 10 is

```
        gcc -O3 -std=gnu11 -Wall -c main.c
```

Once you have done this, you can just `make` it from the command line.

```
node00:~/snsbook/code/column/make$ make
gcc -O3 -std=gnu11 -Wall -c main.c
gcc -O3 -std=gnu11 -Wall -c skel.c
gcc -O3 -std=gnu11 -Wall -o main main.o skel.o
node00:~/snsbook/code/column/make$ ls
Makefile        main.c          skel.c          skel.o
main*           main.o          skel.h
```

You will obtain `main`. You can do the same thing by specifying arguments explicitly.

```
node00:~/snsbook/code/column/make$ make main
```

What is great about `make` is that if you modify a file, only the files that depend on it will be recompiled. For example, if you modify

```
node00:~/snsbook/code/column/make$ touch main.c
node00:~/snsbook/code/column/make$ make
gcc -O3 -std=gnu11 -Wall -c main.c
gcc -O3 -std=gnu11 -Wall -o main main.o skel.o
```

If you artificially update the timestamp of `main.c`, only `main.c` will be recompiled in the next make. `skel.c` and `skel.h` will not be recompiled. Similarly, the

```
node00:~/snsbook/code/column/make$ touch skel.h
node00:~/snsbook/code/column/make$ make
gcc -O3 -std=gnu11 -Wall -c main.c
gcc -O3 -std=gnu11 -Wall -c skel.c
gcc -O3 -std=gnu11 -Wall -o main main.o skel.o
```

If you update the timestamp of `skel.h`, both `main.c` and `skel.c` both will be recompiled because they both depend on each other.

Finally,

```
node00:~/snsbook/code/column/make$ make clean
rm -f main *.o
```

and include instructions for easy cleaning (lines 15, 16).

By the way, `main.o`, `skel.o` is redundant because it repeats the same instructions. It is better to write them in a simpler notation like the following.

Listing 3.13 `Makefile`

```
 1  CC = gcc
 2  CFLAGS = -O3 -std=gnu11 -Wall
 3
 4  all: main
 5
 6  main: main.o skel.o
 7          $(CC) $(CFLAGS) -o $@ $^
 8
 9  main.o skel.o: skel.h
10
11  .c.o:
12          $(CC) $(CFLAGS) -c $<
13
14  clean:
15          rm -f main *.o
```

Line 9 is `main.o`, `skel.o` depends on `skel.h` as usual. Lines 11 and 12 are a new notation, called the **suffix rule**.

Listing 3.14 `Makefile`

```
11  .c.o:
12          $(CC) $(CFLAGS) -c $<
```

`.c` extension means to generate `.o`.

`make` is the first step in the process. Currently, there are various build tools such as `autoconf` and `cmake`. Please find a good tool and use it.

Part II

Reproducing Various Brain Dynamics and Functions

In Part I, we learned the basics of neuroscience and numerical computation, and introduced the mathematical models of neurons and synapses necessary for actual neural circuit simulation, as well as their numerical solution methods. At the end of Part I, we were able to construct and simulate the behavior of a network of LIF model neurons randomly coupled with exponentially decaying synapses. We are now ready to start simulating neural circuits with a network (**spiking network**) using a model of **spiking neurons**. On the other hand, this is just an exercise and a benchmark, so it is not very interesting from a neuroscience point of view at the moment! It's not so interesting from a neuroscientific point of view.

In Part II, we will attempt to reproduce phenomena that are actually seen in the brain using spiking networks. In particular, in some examples, we implement **synaptic plasticity**, which modulates the strength of synaptic connections in response to neuronal activity, and thereby simulate **learning** in the brain. Learning is the source of the brain's flexible and robust information processing mechanisms. Since there are various forms of synaptic plasticity, and existing simulators only provide a part of them, it is significant to be able to develop the code from scratch by yourself.

Furthermore, brain models have been used not only to reproduce and predict phenomena, but also for various engineering applications. Some examples of such applications will be presented. In particular, in recent years, in the context of **neuromorphic computing**, the low power consumption of the brain's information processing mechanism, which is said to be about 20 W, has been in the spotlight again. The core of this is spiking neurons, which is the propagation of spikes.

In Part II, in order to function as a reading material, I will first introduce some interesting phenomena of the brain, and then try to run a simulation to reproduce them. I hope that the reader will be able to appreciate the mystery of the brain itself and the fun of neuroscience research as well as the simulations.

Chapter 4

What is the brain?

First, let's look briefly at the brain itself.

4.1 Brain structure and function

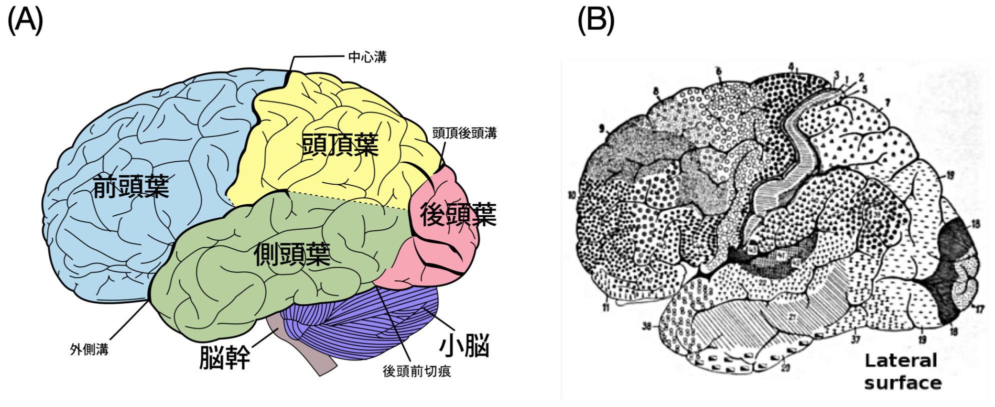


Fig. 4.1 Structure of the brain. (A) The cerebrum is divided into four lobes by deep sulci (central sulcus, lateral sulcus, and parieto-occipital sulcus). (B) Map of Brodmann. Both images taken from Wikipedia [*1].

The **brain** is an organ that resides in our skull (Figure 4.1A). The brain can be divided into four major regions: the **cerebrum**, **cerebellum**, **diencephalon**, and **brainstem** (Kandel et al., 2021). The cerebrum is divided into the **cerebral cortex**, the underlying **white matter**, and the **basal ganglia**, **amygdala**, and **hippocampus** (called the **subcortical structures**). When we think of the brain, we usually think of the cerebrum, which in most cases is the cerebral cortex. The cerebral cortex is dense with neurons and appears grayish white, while the white matter is dense with axon fibers and appears white. The cerebellum is literally a small brain located in the back of the head and connected to the brainstem, but about 80entire brain reside in the cerebellum (Azevedo et al., 2009). The diencephalon consist of the **thalamus** and **hypothalamus**.

The thalamus, in particular, is thought to be a relay point for transmitting information from the body to the cerebral cortex or for interconnecting the cerebral cortex with the basal ganglia and cerebellum. The brainstem is further divided into the **midbrain**, the **pons**, and the **medulla**. The midbrain contains the dopamine-producing **substantia nigra**, which is thought to generate signals for reinforcement learning in the basal ganglia (Chapter 7). The pons is also a relay point

[*1] `https://ja.wikipedia.org/wiki/脳#/media/ファイル:Brain_diagram_ja.svg`, `https://commons.wikimedia.org/wiki/File:Brodmann-areas.png`

for transmitting signals from the cerebral cortex to the cerepons.

4.2 Brain learning and synaptic plasticity

Perhaps the most important factor that makes the brain a brain is the ability to change its own behavior through **learning**. It is thought that the brain has survived by adapting itself to changes in the environment.

Learning in neural circuits is the process of changing the strength of the connections between neurons. In other words, memories are stored in synapses, and by changing the strength of the synaptic connections, memories are rewritten. This mechanism is called **synaptic plasticity**.

4.2.1 Hebb rule — Neurons that fire together, wire together

The basic mechanism of synaptic plasticity is known as the **Hebbian rule** or **Hebbian learning**, proposed by Donald O. Hebb (Hebb, 1949), which states that if two synaptically connected neurons fire simultaneously, the connection between them is strengthened (Figure 4.2).

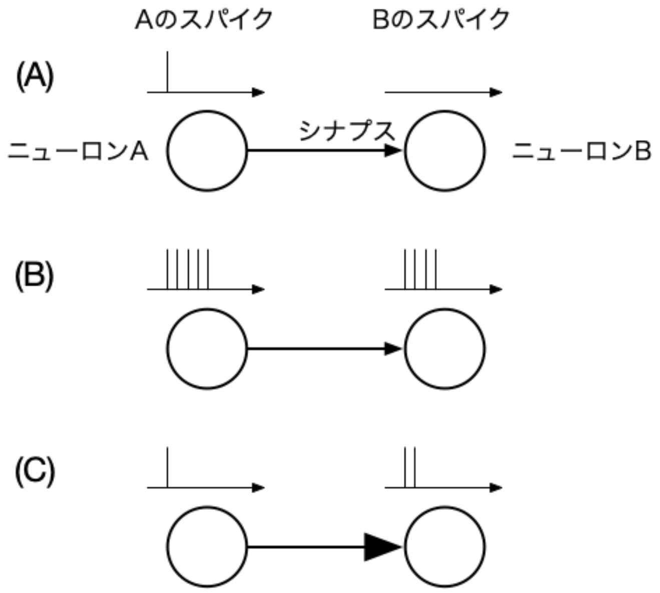


Fig. 4.2 Schematic of the Hebbian rule. (A) (Before learning) A single spike in neuron A does not provide enough excitatory input for neuron B to fire a spike. (B) (During learning) When neuron A fires frequently, neuron B fires spikes as well, and both A and B fire spikes simultaneously. (C) (After learning) The Hebbian rule strengthens the synaptic connections between A and B, and neuron B is able to fire a spike in response to a single spike by neuron A. The size of the arrow indicates the strength of the synaptic connections.

Although Hebb himself only proposed this learning rule as a concept, a similar phenomenon (**long-term potentiation**, **LTP**) was discovered in actual neural circuits (Bliss and Lomo, 1973), and the important role of NMDA receptors at synapses as a mechanism was established (Bliss and Collingridge, 1993).

In practice, the Hebbian rule alone only strengthens connections unilaterally, so a mechanism for weakening connections is also necessary to ensure stable learning. Various methods have been proposed, such as letting it decay spontaneously over time, weakening it when neurons are not firing at the same time (the **anti-Hebb rule**), or more directly normalizing it so that the sum of the strength of the synaptic connections is constant.

4.2.2 Spike timing-dependent plasticity

The Hebbian rule is a concept based on the firing frequency of a neuron, which has been extended to individual spikes in **spike timing-dependent plasticity** (**STDP**) (Caporale and Dan, 2008). The increase or decrease of the coupling strength is determined by the timing of the spike firing of the post-side neuron and the post-side neuron. The specific learning rules and formulas will be introduced in Section 5.3.2.

4.3 Types of brain learning

There are three main ways in which the brain learns.

- **Unsupervised learning**
  Unsupervised learning is the process of extracting and clustering statistical properties contained in input data. It is also called **self-organization**. Unlike supervised learning, which will be explained next, it is not possible to explicitly specify what is to be acquired or represented by learning.
- **Supervised learning**
  In supervised learning, several pairs of contextual and supervisory signals are given as training data, and the system learns to return the same output as the corresponding supervisory signal for each contextual signal input. For example, if the names of various fruits and their colors are trained as the context signal and the teacher signal respectively, after training, the system will output the color of the fruit based only on the name of the fruit. The **perceptron** (Rosenblatt, 1958) , the originator of so-called neural networks, is a neural network that uses supervised learning to achieve this kind of pattern recognition. This was to be done. The perceptron has gained wide consensus as a model for the cerebellum (Section 6), which will be discussed later.
- **Reinforcement learning**
  Reinforcement learning is learning to find an appropriate output for a given contextual signal by trial and error. In this process, a reward signal is given to evaluate whether the output was appropriate or not, and the system learns to maximize the expected value of the reward that will be obtained in the future (Sutton and Barto, 2018). Reinforcement learning is also at the core of Alpha Go (Silver et al., 2016), the world's first game to defeat the world Go champion, and is one of the most important recent topics in the field of machine learning.

Doya proposed that the cerebral cortex, cerebellum, and basal ganglia are responsible for unsupervised learning, supervised learning, and reinforcement learning, respectively (Doya, 1999, 2000a), and it is assumed that the three brain regions use different learning methods to enable robust and flexible whole-brain learning. On the other hand, it is still unclear how these three brain regions work together to perform various tasks, as it is still difficult to measure multiple brain regions simultaneously on a large scale. It is expected that research on learning algorithms in the whole brain will progress in the future (Yamazaki and Lennon, 2019; Yamazaki, 2021).

Chapter 5

Cerebral cortex — Simulation of ocular dominance map formation in the primary visual cortex

The first area to be introduced is the cerebral cortex, which is the most familiar. In sensory information processing, neurons have the property of responding selectively to specific stimuli, and it is known that this property can be acquired. It is known that this property can be acquired. The acquisition process is reproduced by spiking networks.

5.1 Structure and function of the cerebral cortex

The **cerebral cortex** is divided into **frontal**, **parietal**, **occipital**, and **temporal lobes** according to the deep sulci on the surface (Figure 4.1). Each brain lobe has a different function. The frontal lobe is mainly responsible for working memory, motor planning, motor control, and language; the parietal lobe for somatosensory perception and various sensory associations; the occipital lobe for vision; and the temporal lobe for hearing, language, and memory. Figure 4.1B), and the structures generally correspond to differences in function.

Despite the differences in function between locations, the neural circuits of the cortex are almost equally structured in six layers (Figure 5.1A). Layer I, closest to the surface, contains the dendrites of layer II/III neurons and axons from other areas of the cortex. layer II/III consists mainly of small pyramidal neurons that receive signals from neurons in other areas of the cortex and form horizontal connections between the cortex. layer IV consists of small spherical neurons. Layer IV consists of small spheroidal neurons, whose primary input signal is sensory input from the thalamus. Layer V/VI consists of large pyramidal neurons that send signals to other cortical areas and subcortical structures. The basic signal flow is layer IV $\rightarrow$ layer II/III $\rightarrow$ layer V/VI, with input from other cortical areas to layer II/III (Figure 5.1B).

5.2 Learning in the cerebral cortex

In the cerebral cortex, special inputs such as supervisory signals in supervised learning and reward prediction signals in reinforcement learning, represented by climbing fiber inputs in the cerebellum and dopamine inputs in the basal ganglia (see below), are either absent or relatively weak. Therefore, learning that takes place in the cortex is considered to be unsupervised learning [*1]. This is especially true for the learning of sensory information, which has been studied extensively, especially in the

[*1] `https://en.wikipedia.org/wiki/Cerebral_cortex#/media/File:Cajal_cortex_drawings.png`

[*1] Some of the signals are related to dopamine projection and reward, so their involvement has been pointed out.

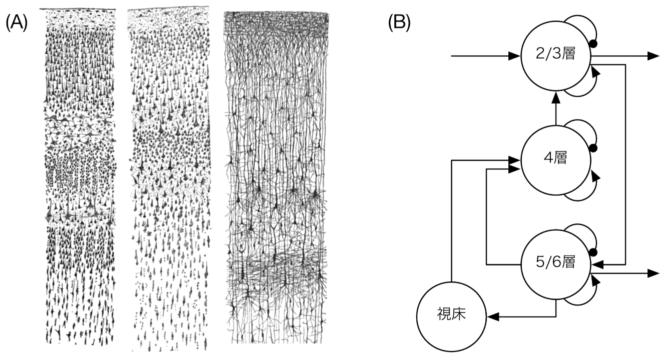


Fig. 5.1 Six-layer structure of the cerebral cortex. (A) Staining images. Different staining methods produce different results (left and center: Nissle staining, right: Golgi staining). Images were taken from Wikipedia [*1]. (B) Schematic diagram of signal flow. Arrows and circles represent excitatory and inhibitory synaptic connections, respectively, and lines and curves represent inter- and intralaminar connections, respectively. Based on Bosman and Aboitiz (2015).

primary visual cortex.

Layer IV of the cerebral cortex is the first place where sensory information from the body enters, especially in the primary visual cortex (V1), where visual information is first received. In particular, it is thought to extract statistical features in visual signals.

Neurons in layer IV respond selectively to inputs from either the right eye or the left eye, which is called **ocular dominance**. In addition, neurons that are close to each other in the cortex have the same ocular dominance, and it is known that right- and left-eye dominant areas form a stripe or spot pattern on the cortical surface. This spatial structure is called the **ocular dominance map** or **ocular dominance column**.

It is thought that the ocular dominance map is essentially formed by postnatal visual experiences [*2]. For example, it is known experimentally that if one eye is blinded immediately after birth, the number of neurons that respond to input from the unseen eye becomes extremely low (Shatz and Stryker, 1978).

Eye dominance is a property produced by changes in the strength of synaptic connections between inputs from the thalamus and neurons in layer IV. layer IV neurons receive inputs from both the right and left eyes in the early postnatal period, but only one of the connections is thought to survive through visual experience.

In V1, the left and right inputs are separated in this way, making it possible to detect binocular disparity. In the secondary visual cortex (V2), the proportion of neurons that respond to binocular stimuli increases. As we move to higher levels of the visual cortex, experiments have shown that depth perception and stereopsis using binocular disparity are possible.

5.3 Simulation of ocular dominance map formation

Let's build a spiking network that reproduces the formation of the eye dominance map.

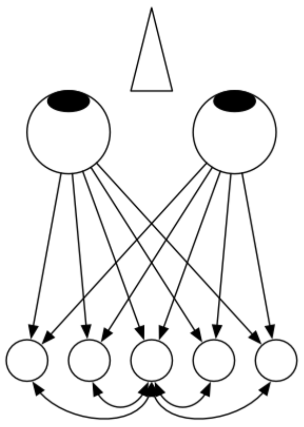


Fig. 5.2 Overview of the eye dominance map formation model

5.3.1 Model

Our reference is the classical model of ocular dominance map formation (Miller et al., 1989) (Figure 5.2). We prepare one spiking neuron corresponding to each photoreceptor, which is a light-responsive cell in the retina of the left and right eyes, and each of them is assumed to fire spikes independently. In the simulation program, we assume a Poisson spike of 100 spikes/s (Section 3.3.2) and generate a spike (0 or 1) with a random number every Δt (= 1 ms) and store it in the following variables.

$$\texttt{s\_eye[}X\texttt{]} = \begin{cases} 1 & Spikes fired at that time \\ 0 & otherwise \end{cases} \tag{5.1}$$

where X is one of the photoreceptors of the left eye (`L`) or the right eye (`R`).

The visual cortex is considered a two-dimensional plane, and 32 × 32 neurons are set up in the simulation program. The input to neuron i in the visual cortex consists of spike input aff(i) from photoreceptors and spike input lat(i) from other neurons via horizontal connections in the cortex. The former is the sum of the product of the spikes of photoreceptor cells `L` and `R` and their coupling strength $w_{i,\texttt{L}} and w_{i,\texttt{R}}$, and is expressed as

$$\mathrm{aff}(i) = w_{i,\texttt{L}} \cdot \texttt{s\_eye[L]} + w_{i,\texttt{R}} \cdot \texttt{s\_eye[R]} \tag{5.2}$$

The latter is calculated by defining the spike (0 or 1) of neuron j in the visual cortex by the variable `s_ctx [ j ]` and the function of horizontal coupling between neurons i and j by $I(i, j)$, respectively, and using the following equation

$$\mathrm{lat}(i) = \sum_j I(i, j) \cdot \texttt{s\_ctx [j]} \tag{5.3}$$

For simplicity's sake, let's assume that the strength of horizontal connectivity depends on the distance between neurons, and that it is a Mexican hat-type function with excitatory connectivity dominance for short distances and inhibitory connectivity dominance for long distances (Figure 5.3).

$$I(i, j) = k_1 \exp\left(-\frac{|i-j|^2}{2\sigma_1^2}\right) - k_2 \exp\left(-\frac{|i-j|^2}{2\sigma_2^2}\right) \tag{5.4}$$

*2 The basic structure of a species is thought to be genetically determined. How much of brain development is inborn and how much is acquired is one of the biggest debates in the history of neuroscience.

where $|i-j|$ is the distance between neurons i and j in the two-dimensional plane of the cortex [*3].

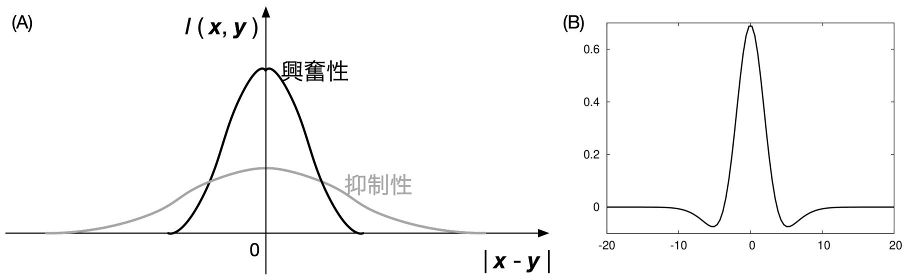


Fig. 5.3 Shape of horizontal connections. The horizontal axis represents the distance between neurons and the vertical axis represents the coupling strength. (A) Coupling through inhibitory neurons is expected to be more distant than excitatory coupling, because the inhibitory neurons are expected to reach a greater distance than the excitatory neurons. Increase the width of the inhibitory bonds. (B) Plot of equation (5.4). The parameters are $\sigma_1^2 = 4.0$, $\sigma_2^2 = 13.0$, $k_1 = 1.0$, $k_2 = \sigma_1^2/\sigma_2^2$. The negative coupling strength means that the inhibitory input comes indirectly through the inhibitory neurons. In this case, it is a two-dimensional function.

The currents g_{aff} and g_{lat} are calculated from these two values, and the membrane potential is updated and spike firing is performed. The initial value of synaptic coupling strength is randomly determined in the range of 0.5 ± 0.1, assuming equal coupling from left and right. The current-based LIF model (Section 3.2 節) and the exponential decay model (Section 3.4) are employed for neurons and synapses, respectively.

This alone will work as a network, but it does not introduce the rules of synaptic plasticity, so naturally no eye dominance map will be formed.

5.3.2 Synaptic plasticity

In this section, we will introduce synaptic plasticity, and discuss STDP, which was mentioned in section 4.2.

The concept of STDP is very simple. Consider the timing of the firing of the pre-side neuron and the firing of the post-side neuron. If the pre-side firing occurs slightly before the post-side firing, the coupling is strengthened. This means that the coupling becomes stronger when there is a firing-order relationship in which a spike input from the pre-side causes the post-side to fire. Conversely, if the post-side fires slightly before the pre-side, the coupling is weakened. This means that the coupling is weakened when there is a relationship in the firing order where the post-side firing is not caused by the pre-side firing (Figure 5.4). Here, the original Hebbian rule only considered strengthening the coupling, but it generally includes extensions in the direction of weakening, and this STDP rule also incorporates both strengthening and weakening.

There are many variations of STDP, but we consider the simplest implementation here (see Chapter 19.2.2 in Gerstner et al. (2014) for more details). First, we prepare the value of the trace, where i and j are the indices of the post- and pre-side neurons, respectively, and consider the synaptic coupling strength w_{ij} between them. We define the pre-side trace x_j as follows:

$$x_j(t) = \exp\left(-\frac{\Delta t}{\tau_{\mathrm{pre}}}\right) x_j(t-\Delta t) + S_j(t) \tag{5.5}$$

where τ_{pre} is a decay time constant, $S_j(t)$ is 1 when the neuron fires a spike at time t, and 0 otherwise. The form of the equation is the same as in the calculation of the exponential decay

[*3]Note that it is not the absolute value of the difference between integers i, j

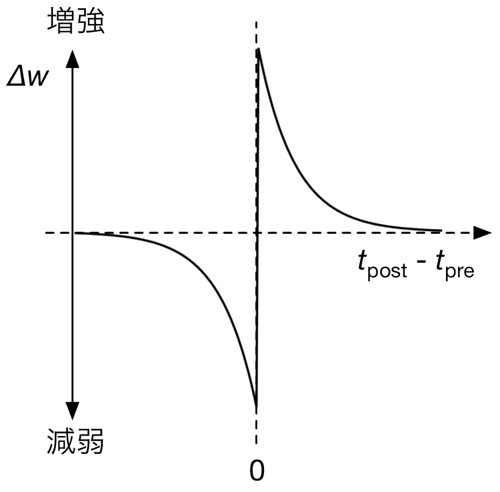


Fig. 5.4 Window function of STDP. The horizontal axis is the time difference between the pre and post firings, and the vertical axis is the amount of change in coupling strength.

synapse (Equation (3.16)). The trace increases in value when it fires a spike, and then decays exponentially. Similarly, consider the trace y_i on the post side.

$$y_i(t) = \exp\left(-\frac{\Delta t}{\tau_{\mathrm{post}}}\right) y_i(t - \Delta t) + S_i(t) \tag{5.6}$$

τ_{post} is the time constant of decay. Note that the trace represents an exponential window of the STDP. In other words, the time constant defines the shape of the window function in STDP. We use this value to define the coupling strength as

$$\begin{aligned} w_{ij}(t + \Delta t) &= w_{ij}(t) + \Delta w_{ij}(t), \\ \Delta w_{ij}(t) &= -A_- y_i(t) S_j(t) + A_+ x_j(t) S_i(t) \end{aligned} \tag{5.7}$$

and update. Here, A_- and A_+ are constants. Simply put, for each spike of the pre-side, the 1 value of the trace of the post-side at that time is subtracted, and conversely, for each spike of the post-side, the 1 value of the trace of the pre-side at that time is added.

Note that there is no specific supervisory signal that dictates ocular dominance in this model; STDP is expected to automatically form ocular dominance in a self-organizing manner. Therefore, this falls into the category of unsupervised learning.

5.3.3 Simulation results

Let's run a simulation in which 32 × 32 V1 neurons independently receive inputs from the right and left eyes as Poisson spikes of 100 spikes/s, and also receive spikes from other nearby neurons via Mexican hat-shaped horizontal connections. STDP alters the strength of synaptic connections between photoreceptors and neurons in V1.

The source code is located in `code/part2/od/`. Copy the entire folder and `make` to create `od`. When executed, `before.dat` and `after.dat` will be generated.

```
node00:~/snsbook/code/part2/od$ make
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1/ -DSFMT_MEXP=19937 -o SFMT.o -c ../misc/SFMT-
    src-1.5.1/SFMT.c
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1/ -DSFMT_MEXP=19937 -o od od.c SFMT.o -lm
node00:~/snsbook/code/part2/od$ ./od
```

```
node00:~/snsbook/code/part2/od$ ls
Makefile        SFMT.o          after.dat       before.dat      od*             od.c
    work/
```

`before.dat`, `after.dat` are the eye dominance of V1 neurons before and after learning, respectively. We defined the eye dominance of neuron i as $w_i^{\mathrm{od}} = w_{i,L} - w_{i,R}$. It takes values in the range $[-1, +1]$, where negative values indicate right eye dominance and positive values indicate left eye dominance. Each file can be visualized using gnuplot as follows.

```
1 node00:~/snsbook/code/part2/od$ gnuplot
2 :
3 gnuplot> set palette defined (-1 "blue", 0 "white", 1 "red")
4 gnuplot> set pm3d map
5 gnuplot> set size square
6 gnuplot> set cbrange [-1:1]
7 gnuplot> set xrange [0:31]
8 gnuplot> set yrange [0:31]
9 gnuplot> splot 'before.dat'
```

What we want to do is to display the values assigned to each point on a 32×32 two-dimensional plane. What we want to do is to display a bird's eye view of the values assigned to each point on a 32×32 two-dimensional plane. line 8 creates a bird's eye view of the plane, and line 9 makes the plane square. line 10 sets the range of values to be displayed from -1 to 1, and lines 11 and 12 set the ranges of the x-axis and y-axis, respectively. Display the data in line 13. Note that the color palette should be set appropriately to make it easier to see the sign of the values (line 7). Here, we use white as the standard, blue for negative (right eye dominant) and red for positive (left eye dominant). bef or e. dat is now displayed almost exclusively in white. I think the display of `before.dat` is almost white, because the initial bond strength is small. How about the display of `after.dat`? The result is shown in Figure 5.5. We can see that the positive and negative regions appear like stripes, forming a pattern similar to the ocular dominance map seen in V1.

5.3.4 Simulation of monocular deprivation

Next, let's try to simulate monocular shielding. If the animal is raised with one eye occluded, the area of the stripe pattern corresponding to the open eye will be larger. In this simulation, the firing frequency of the right eye and the left eye is assumed to be equal, 100 spikes/s. However, we can add a gradient, such as 50 spikes/s for the right eye and 100 spikes/s for the left eye. The corresponding map will be formed (Figure 5.6). By decreasing the firing frequency on one side, the input on the opposite side becomes dominant, and more neurons selectively respond to the dominant stimulus.

The width of this stripe pattern is determined by the horizontal coupling of the Mexican hat pattern. Specifically, this function is transformed into a Fourier transform, and the spatial frequency that takes the maximum value becomes the width. In other words, this model can be used to calculate the spatial width of a two-dimensional noise. It corresponds to applying a band-pass filter (Swindale, 1996). Thus, by controlling the strength of intracortical inhibition, the width of the stripes can also be modified (Hensch and Stryker, 2004).

Finally, the correlation between the right eye and left eye inputs is important in the eye dominance map. In Figure 5.5, there is a complete lack of correlation, so a striped pattern is formed in which the right eye dominance and left eye dominance are tightly separated. This pattern formation based on the **correlation of inputs** is considered to be the essence of self-organization in V1 (Tanaka, 1990). It is also known that V1 neurons are selective for light segments or edges that are tilted at a specific angle, and this property is called **orientation selectivity**, and the angle that causes the strongest activity is called the optimal orientation. In addition, in the cortex, neurons with similar orientation selectivity are spatially located close to each other, forming a map in which the optimal orientation changes gradually over the cortex. The formation of such orientation-selective maps can

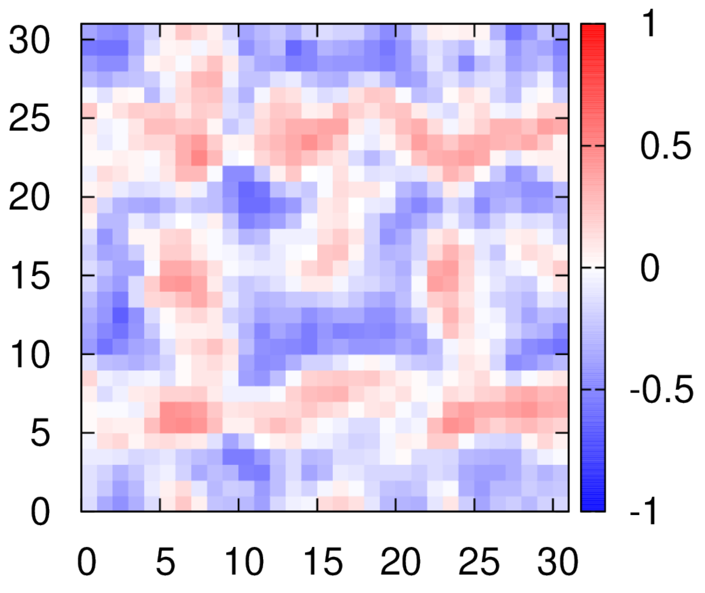


Fig. 5.5 Simulation of ocular dominance map formation. Blue represents right eye dominance and red represents left eye dominance.

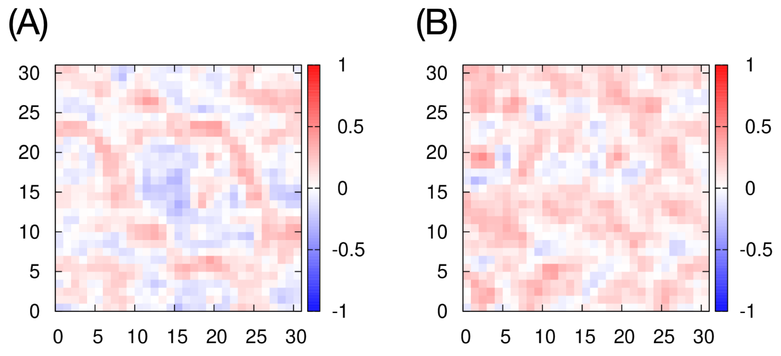


Fig. 5.6 単眼遮蔽のシミュレーション。(A) 右目の視細胞の発火頻度を 1/2 にした場合。(B) 右目の視細胞の発火頻度を 1/10 にした場合。図の見方は図 5.5 と同じ。

be explained by the correlation between the activity of on-type photoreceptor cells that respond to bright spotlight and off-type photoreceptor cells that respond to dark spotlight (Miyashita and Tanaka, 1992; Miller, 1994). The mechanism of V1 self-organization was very well understood in the 1990s, supported by this solid theory and the development of imaging techniques for endogenous signals.

Chapter 6

Cerebellum — Simulation of eyeblink conditioning

After the cerebral cortex, the cerebellum is next. In particular, we will use the spiking network of the cerebellum to reproduce how the cerebellum learns the timing of movements, which is essential for precise motor control.

6.1 Cerebellar structure and function

The **cerebellum** is located in the back of the head and is connected to the brain stem. The cerebellum is located in the back of the head and is connected to the brainstem. Traditionally, the cerebellum has been known to play an important role in motor control and motor learning, and damage to the cerebellum can cause a variety of motor ataxias. In recent years, anatomical findings of interconnections with the cerebral cortex and fMRI imaging studies have suggested that the cerebellum is involved not only in motor functions but also in cognitive functions (Ito, 2012).

The cerebellum consists of three parts: the **cerebellar cortex**, which is the surface of the brain and contains many neurons, the white matter inside the cortex, and the **deep cerebellar nuclei**, which receive the output of the cerebellar cortex. The cerebellum is horizontally divided into regions called **lobes**, with numerous fine wrinkles running horizontally. In the anterior-posterior direction, the cerebellum consists of a central, vertically raised area called the **vermis**, and hemispheres on either side of it. The hemisphere is further divided into the inner and outer hemispheres (Figure 6.1).

The cerebellum is functionally classified into **vestibulocerebellum**, **spinocerebellum**, and **cerebrocerebellum**. The vestibulocerebellum is embryologically the oldest part of the vertebrate brain and is also present in fish. The vestibular cerebellum is embryologically the oldest part of the vertebrate body and is also present in fish, where it is responsible for reflexive eye movements and postural control. In particular, the region known as the hemisphere is responsible for gain adaptation of the **vestibuloocular reflex** (**VOR**) and has played a central role in investigating the relationship between the cerebellum and motor learning. The spinocerebellum consists of the insect part and the inner part of the adjacent hemisphere. The cerebellar cerebellum is responsible for voluntary eye movements, postural control, and gait, and in particular, the cerebellar portion has been studied in depth for adaptation to voluntary eye movements called **saccades** (rapid eye movements).

The cerebrocerebellum is the outer part of the hemisphere and receives input from the cerebral cortex via the bridge and outputs to the cerebral cortex via the thalamus (**cerebro-cerebellar loop**). The cerebellum is the largest and most developed part of the human brain, and is thought to be responsible for the planning and coordination of voluntary movements, while the outermost part of the cerebellum is connected to the prefrontal cortex and is thought to be involved in cognitive functions such as working memory. The cerebrocerebellum is the largest and most developed part of the brain in humans, and has been reported to correlate with symptoms such as autism (Tsai et al.,

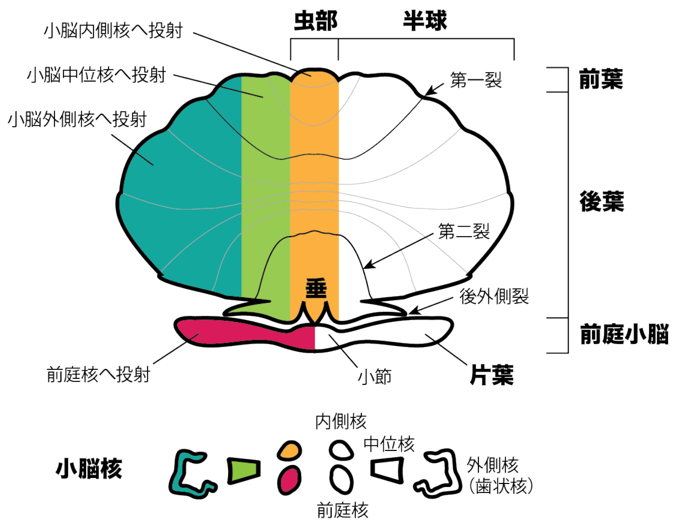


Fig. 6.1 Structure of the cerebellum. The image was newly created based on Figure 2 of the Brain Science Dictionary.

2012).

The circuitry of the cerebellum is much clearer than that of the cerebral cortex.

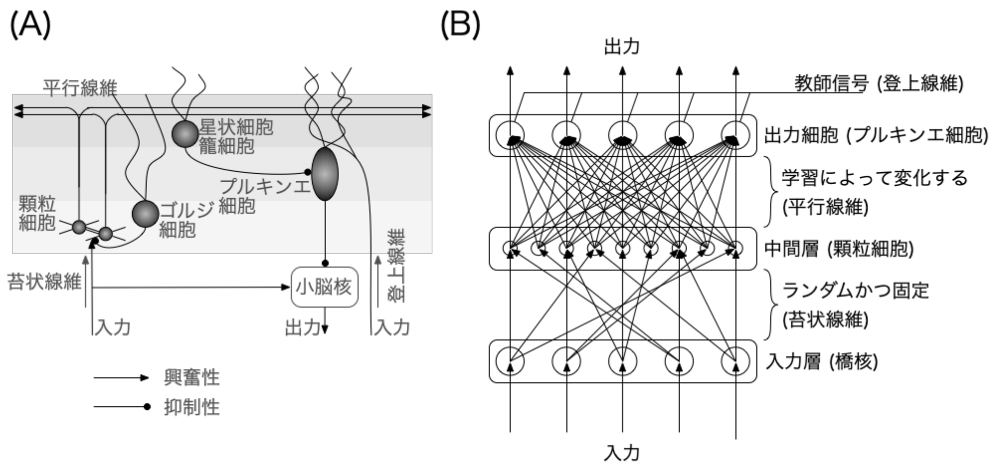


Fig. 6.2 Neural circuits of the cerebellum. (A) Schematic of a microcomplex. (B) Perceptron circuit corresponding to a microcomplex.

The cerebellum is a two-input, one-output circuit (Figure 6.2A). One input is **mossy fibers** that provide contextual signals, which enter the cerebellar cortex via bridges. The mossy fibers provide excitatory input to the **granule cells**. Granule cell activity excites **Purkinje cells**, **molecular layer interneurons** (**stellate cells** and **basket cells**), and **Golgi cells** via their axons, the **parallel fibers**. Purkinje cells are the final output of the cerebellar cortex and inhibit the deep cerebellar nuclei. Molecular layer interneurons and Golgi cells inhibit Purkinje cells and granule cells, respectively. The deep cerebellar nuclei receive excitatory input from mossy fibers in addition to inhibitory input from Purkinje cells. Another input to the cerebellum is the **climbing fibers**, which provide a teacher signal from the **inferior olive**, projecting to Purkinje cells, strongly depolarizing them, and inducing synaptic plasticity between the parallel fiber-Purkinje cell synapses. The output of the deep cerebellar nuclei then becomes the final output of the cerebellum. This circuit is a functional unit and is named the cerebellar **corticonuclearmicrocomplex**. The circuitry of this microcomplex is repeated in the cerebellum in a copy-and-paste fashion, forming a circuit that resembles a very orderly crystalline structure.

The cerebellum plays an important role in motor control and motor learning, and damage to the cerebellum results in inability to move properly. Typical cerebellar symptoms include decreased muscle tone, impaired coordination, dysmetria, nystagmus, and tremor. However, motor activity itself does not disappear, only the tone of movement (ataxia). This is in contrast to damage to the cerebral cortex, which results in loss of movement when the motor cortex is damaged. In rare cases, humans are born without a cerebellum (cerebellar agenesis), and these individuals are still able to control their movements to the extent that they can ride a bicycle, albeit with some sluggishness (Glickstein, 1994). This suggests that the motor memory retained in the cerebellum is acquired and can be managed to some extent without the cerebellum.

6.2 Cerebellar learning

David C. Marr and James S. Albus independently proposed that the microcomplex is a perceptron and the cerebellum is a supervised learning machine, viewing the mossy fiber input as the context signal, the granule cells as the middle layer, the Purkinje cells as the output layer, and the climbing fiber input as the supervised signal (Figure 6.2) (Marr, 1969; Albus, 1971). In particular, it is important to note that memory is stored in the synapses between parallel fibers and Purkinje cells. Masao Ito applied this idea to the vestibular oculomotor reflex, a reflex adaptation of eye movements (Ito, 1970). Ten years later, Ito et al. actually demonstrated that climbing fiber stimulation weakens the strength of parallel fiber synaptic connections (**long-term depression**, **LTD**) (Ito et al., 1982) and established the theory that is now known as the **Marr-Albus-Ito model**. This model established the direction of cerebellar research and has served as a compass for further research. Furthermore, efforts are continuously being made to develop this model to gain a more accurate understanding of the information processing mechanisms of the cerebellum (Yamazaki and Lennon, 2019; Yamazaki, 2021).

Our daily movements are achieved by contracting all the muscles in the body at the right size and at the right time. The cerebellum is thought to play a role in adaptively controlling the appropriate magnitude (gain) and timing of movements by learning. The former is studied by VOR gain adaptation, and the latter by **eyeblink conditioning**. In this book, we will actually simulate the latter.

6.3 Simulation of eyeblink conditioning

Eyeblink conditioning is a task in which a conditioned stimulus (e.g., a sound) is paired with a nociceptive stimulus (e.g., an air puff on the eyelid) and the sound is played and the air puff is given to induce blinking (an unconditioned response) (Mauk and Donegan, 1997; Christian and Thompson, 2003) (Figure 6.3). After repeated presentation of the stimulus, the animal will blink when it hears the sound, even without the air puff. This means that sound and blinking are conditioned. If we consider the sound and the air puff as a context signal and a teacher signal, respectively, this is supervised learning. If the time between the sound and the air puff is fixed at, say, 250 ms, the subject will be conditioned to blink 250 ms after the sound. In other words, not only are the sound and blink conditioned, but the passage of 250 ms is represented somewhere in the brain, and the timing of the blink is also learned. When the time until the air puff is given is changed, the timing of the blink changes accordingly, indicating that the timing of the blink is specified by the air puff. In addition, damage to the cerebellum impairs blink timing, suggesting that the cerebellum plays an important role in this process.

In the brain, sound and air puff stimuli are input to the cerebellar cortex from mossy fibers and

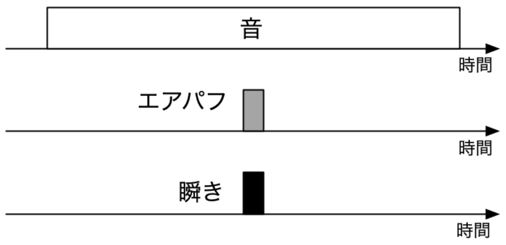


Fig. 6.3 Conceptual diagram of blink reflex conditioning.

climbing fibers, respectively, and Purkinje cells control motor commands for blinking. As mentioned above, the strength of the synaptic connections between parallel fibers and Purkinje cells changes with learning.

The interesting thing about blink reflex conditioning is that it not only conditions the blink to a sound, but also causes the blink to occur at a specified timing. For example, if it is 250 ms, how can the time lapse of 250 ms be represented in the brain? Let's try a simulation.

6.3.1 Simulation results

The code for the simulation can be found at `code/part2/ec/`. This extracts only the essence of blink reflex conditioning, which consists of three types of cells in the cerebellar cortex: granule cells, Golgi cells, and Purkinje cells [*1].

```
node00:~/snsbook/code/part2/ec$ make
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -DSFMT_MEXP=19937 -c main.c
:
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -DSFMT_MEXP=19937 -o main main.o gr.o go.o pc.o
     conn.o SFMT.o -lm
node00:~/snsbook/code/part2/ec$ ./main
trial = 0
:
trial = 49
```

When the execution is finished, the following file will be generated.

- `go.spk.`n: Spike trains of Golgi cells in the n-th trial
- `gr.spk.`n: Spike trains of granule cells in the n-th trial
- `pc.mbp.`n: Membrane potentials of Purkinje cells in the n-th trial
- `pc.spk.`n: Spike trains of Purkinje cells in the n-th trial

First, let's discuss the representation of the passage of time. In this model, the passage of time is represented by the dynamics of granule cells and Golgi cells. Granule cells excite Golgi cells via parallel fibers, and Golgi cells inhibit granule cells in turn. Thus, these cells consitute a **recurrent network**. If this coupling is left random, then in response to the presentation of a sound stimulus, the granule cells will respond to the stimulus as shown in Figure 6.4. firing pattern is shown as It can be plotted as follows.

```
node00:~/snsbook/code/part2/ec$ gnuplot
:
gnuplot> plot 'gr.spk.0' with dots
```

At first glance, the spikes appear to fire randomly, but this firing pattern is generated by random coupling with Golgi cells, which produce almost identical firing patterns when the same sound

[*1]Modified from Yamazaki and Tanaka (2007); Yamazaki and Nagao (2012).

stimulus is presented [*2].

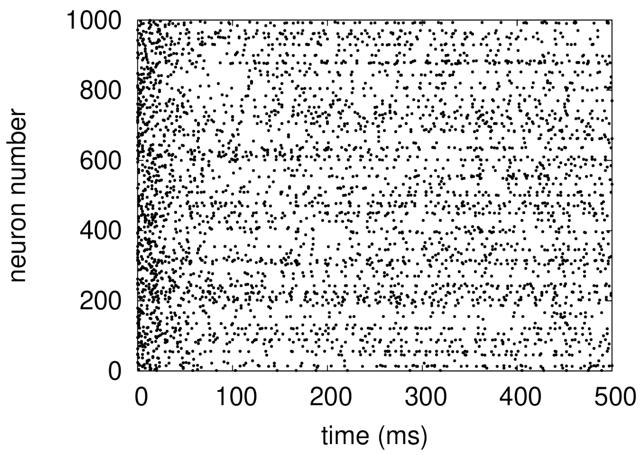


Fig. 6.4 Raster plot of granule cells. Only 1000 neurons are shown. 1 point represents 1 spike. The stimuli were presented at 0 ms for 500 ms.

What is the significance of this seemingly random pattern? We calculated the raster plot and PSTH (Section 3.6.3) for each neuron for 50 trials, and the results for the three neurons with characteristic firing patterns are displayed in Figure 6.5. As can be seen from the raster plot, each neuron shows almost the same firing pattern for each trial. On the other hand, as can be seen from the PSTH, the temporal variation of the firing pattern varies, with some neurons firing periodically every 100 ms (left), some firing only immediately after the presentation of the sound stimulus (center), and some firing strongly immediately after the stimulus and then firing at a constant low frequency (right). Neurons not shown here also show their own firing patterns.

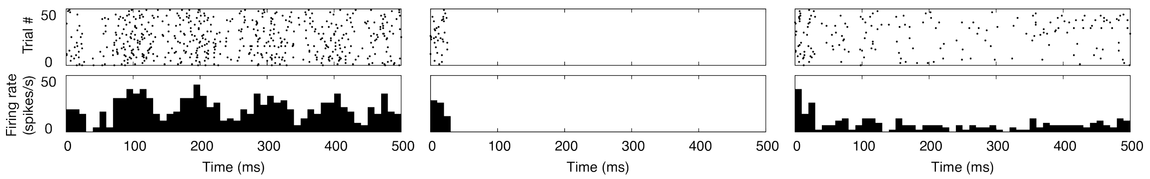


Fig. 6.5 Raster plots of 50 trials of granule cells 2, 3, and 50 (left to right) (top) and their PSTH (bottom). The bin width of the PSTH is 10 ms, and the sound stimulus started at 0 ms and was presented for 500 ms.

The population of granule cells that fire spikes at a given time is determined, but that population does not fire spikes as a group at other times. In addition, the population of spiking granule cells changes little by little from time to time. This means that the passage of time can be represented by the firing of clusters (Yamazaki and Tanaka, 2005). Therefore, the population of granule cells that fire spikes during air puff presentation is uniquely determined, and if only the synaptic connections of that population are weakened, the excitatory input to the Purkinje cell will cease, and the Purkinje cell should stop firing spikes only at that time (Figure 6.6).

The changes in the firing pattern of Purkinje cells are shown in Figure 6.7. In the first trial of conditioning, Purkinje cells only fire spikes at a high frequency. As learning progresses, the firing frequency gradually decreases toward the time of the air puff, and when learning progresses sufficiently, the Purkinje cells completely stop firing spikes before and after the air puff presentation.

[*2] The spike timing of each shot is slightly different each trial.

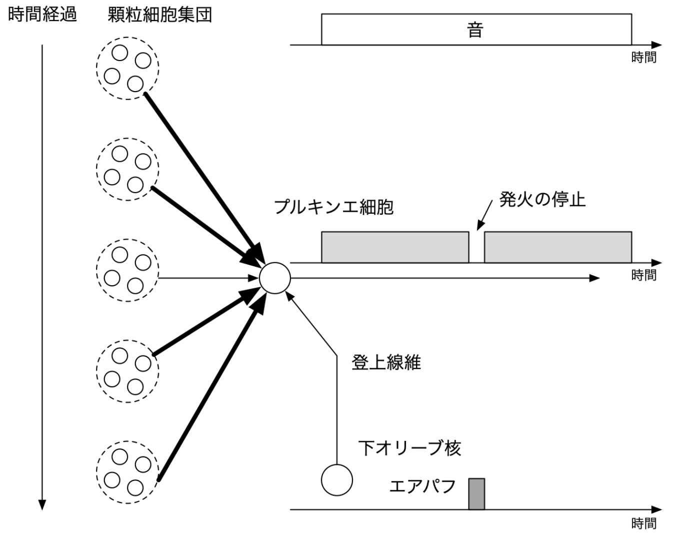


Fig. 6.6 Schematic representation of the behavior of the cerebellar cortex during blink reflex conditioning. From the onset of the sound stimulus, the active granule cell population changes from time to time (dashed circles). The strength of the synaptic connections with Purkinje cells in the population active during air puffs (third from the top) is weakened (thin arrow), while other connections remain unchanged (thick arrow). Thus, the excitatory input to the Purkinje cell decreases before and after the air puff presentation, resulting in the Purkinje cell ceasing to fire.

It is known that when Purkinje cells stop firing, the cerebellar nuclei downstream of the Purkinje cells are deactivated and fire in strong bursts, which drive the eyelid muscles to blink. This is how the blink reflex conditioning is reproduced.

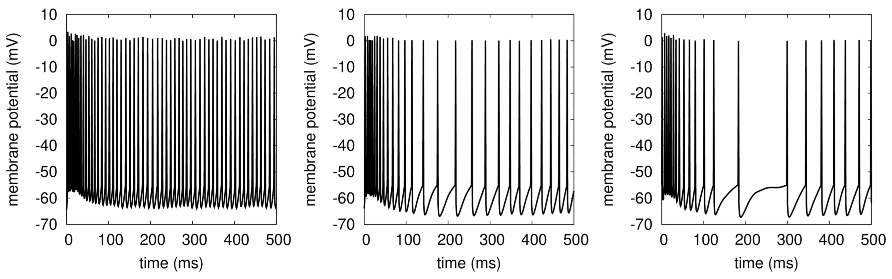


Fig. 6.7 Changes in the firing pattern of Purkinje cells. Air puffs were simulated at 250 ms after the onset of the sound stimulus. The membrane potential of Purkinje cell #10 at trials 1, 10, and 20 from the left.

The membrane potential can be plotted as follows, but note that the vertical bars of the spikes are not shown because it is a LIF model. In fact, in Figure 6.7, the bars are added later and shifted so that the start time of the sound stimulus is 0 ms.

```
node00:~/snsbook/code/part2/ec$ gnuplot
:
gnuplot> set size square
gnuplot> plot [250:750] [-70:10] 'pc.mbp.0' with lines
```

Finally, although this model represents the passage of time by the dynamics of the network of granule and Golgi cells, this representation is one of the hypotheses and has not been fully tested experimentally. It should be noted that this is only one of the various models that have been proposed (Yamazaki, 2021). Thus, models of neural circuit behavior that are not experimentally evident can be tested and predicted, which is a role of spiking network simulation.

Chapter 7

Basal ganglia — Simulation of goal search by reinforcement learning

After unsupervised learning (cerebral cortex) and supervised learning (cerebellum), we will now introduce reinforcement learning. Reinforcement learning is thought to be carried out by the basal ganglia, which acquire appropriate behaviors through trial and error. We will use a reinforcement learner based on a spiking network to perform goal search [*1].

7.1 Structure and function of the basal ganglia

The **basal ganglia** are located in the deep part of the cerebrum. It constitutes a **cortical-basal ganglia loop** that receives input from all parts of the cortex and returns output to the cortex via the thalamus. The basal ganglia are composed of multiple neuronal nuclei. There are **direct paths** that project directly from the **striatum**, the input layer, to the internal segment of the **globus pallidus / reticular formation of the substantia nigra**, the output layer, and **indirect paths** that project via the external segment of the globus pallidus. The striatum also receives **dopamine** projections from the **substantia nigra pars compacta** (Figure 7.1).

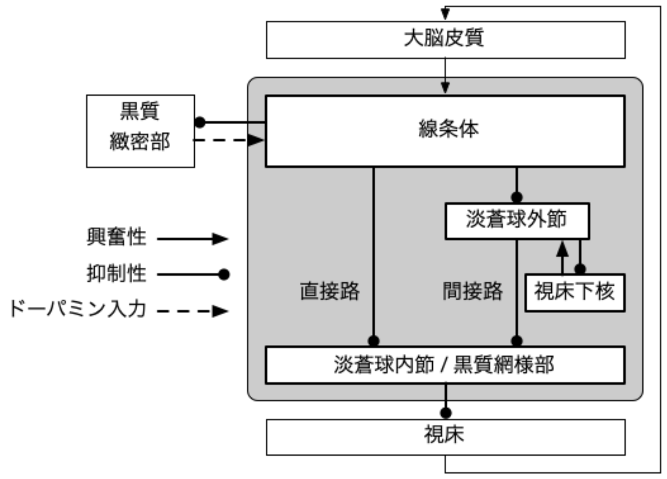


Fig. 7.1 Structure of the basal ganglia

The basal ganglia are thought to play a major role in motor planning, action selection, and decision making. In particular, **Parkinson's disease** is known to be a typical case of basal ganglia disease

[*1] A task called Grid World in the context of reinforcement learning.

caused by dopamine depletion (Jankovic, 2008), strongly suggesting its importance in the control of voluntary movements.

7.2 Learning in the basal ganglia

Experiments with monkeys have shown that the basal ganglia are thought to rely on the neuromodulator dopamine for learning, specifically reinforcement learning (Schultz, 1998). The monkeys were presented with a visual stimulus and asked to pull a lever after a certain period of time to receive a reward, and the activity of dopamine neurons was recorded (Figure 7.2). Before learning, dopamine neurons respond to the reward itself (Figure 7.2A), but after sufficient learning, they respond to visual stimuli (Figure 7.2B). Furthermore, when the reward is deliberately withheld, the activity is rather suppressed at the time when the reward was obtained (Figure 7.2)C). These results suggest that dopamine neurons represent reward prediction (in this case, visual stimuli) rather than reward itself.

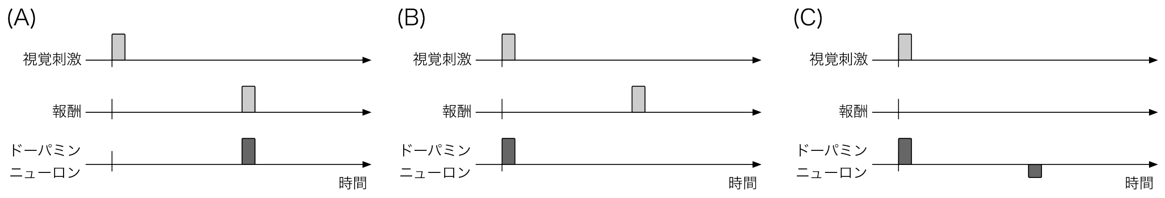


Fig. 7.2 Dopamine neuron activity. (A) Before learning. Dopamine neurons respond to reward. (B) After learning. Dopamine neurons respond to a visual stimulus that anticipates a reward. (C) After learning, when the reward presentation is stopped, dopamine neurons inhibit their activity.

Even more interestingly, this sequence of dopamine neuron activity is what we call reinforcement learning. It was shown to be in good agreement with changes in **temporal difference (TD) error** (see below) (Schultz et al., 1997). These series of studies led to the idea that the basal ganglia perform reinforcement learning using TD errors represented by dopamine neurons, which was later confirmed experimentally (Samejima et al., 2005).

Here, let us briefly summarize reinforcement learning. In particular, we will introduce a reinforcement learning algorithm called the Actor-Critic Method (Sutton and Barto, 2018).

In reinforcement learning, the correct answer itself is not given, unlike supervised learning, but a value called the **reward**, which is an evaluation of how good the action was, is given. This reward value is used to advance learning.

In reinforcement learning, we consider that there is an **environment** and an **agent** placed in that environment. The agent is in a certain **state** in the environment and decides what **action** to take based on its own **policy**. After the action, the agent transitions to the next state and is rewarded for how good the action was. (Figure 7.3A). The agent proceeds with learning based on the principle of **maximizing the expected value of the reward to be obtained in the future**. It is as if an animal gradually acquires an appropriate behavior based on trial and error.

Both the environment and the agent move in discrete time steps $t = 0, 1, 2, \ldots$ At time t, the agent is in state $s_t \in S$, where S is the space of states. The agent takes an action $a_t \in A(s_t)$ based on its own strategy ($A(s_t)$ is the set of possible actions in state s_t). At the next time $t + 1$, the agent receives a reward r_{t+1}, and the state transitions to s_{t+1}. Here, a policy is a probability distribution. The policy at time t is denoted by π_t, tand the probability of taking action a in state s is denoted by $\pi_t(s, a)$. In general, reinforcement learning problems are formulated in discretized time and space, but it is also possible to formulate them in continuous time and space (Doya, 2000b).

The sum of the rewards from time t into the future is called the **return**. Here, the rewards in the

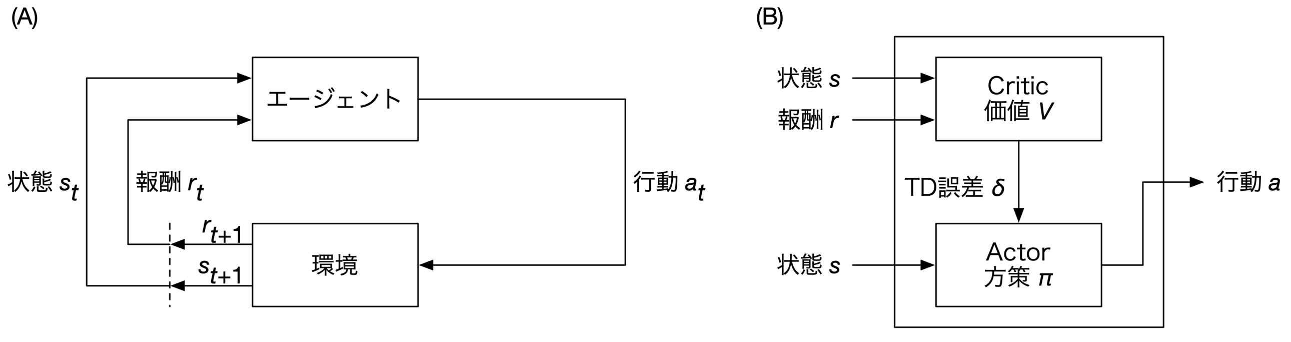


Fig. 7.3 Reinforcement learning problem. (A) Problem setup. (B) Overview of the Actor-Critic method.

distant future are considered to be uncertain compared to the most recent rewards, and are summed over a constant $0 \leq \gamma \leq 1$ In other words, the return R_t is

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+1+i}$$

γ has the name of **discount rate**.

There are many variations of reinforcement learning algorithms, but here we will discuss the Actor-Critic method, in which the agent is divided into an Actor and a Critic (Figure 7.3B). The actor has a strategy π, and the crit ic has a function V (called the **value function**) that represents the **value** of the state (how much return can be obtained by starting from that state). Learning takes place as follows. Suppose an agent in state s performs action a, transitions to state s', and obtains a reward r. First, we calculate a quantity called TD error Δ as follows:

$$\delta = r + \gamma V(s') - V(s) \tag{7.1}$$

Since γ is a constant discount rate and Critic has the value of V, δ is also computed by Critic. V can be updated by learning as follows:

$$V(s) \leftarrow V(s) + \alpha\delta \tag{7.2}$$

where α is a constant for learning rate. On the other hand, the agent calculates the policy π as follows:

$$\pi(s,a) = \frac{\exp(p(s,a))}{\sum_{a'} \exp(p(s,a'))} \tag{7.3}$$

Here, $p(s,a)$ is a value such as the likelihood of action a in state s. If this value is high, the action is more likely to be selected. The method of determining policy, which are probability distributions, as in Equation (7.3) is called the **Softmax method**. $p(s,a)$ is updated by learning as follows.

$$p(s,a) \leftarrow p(s,a) + \beta\delta \tag{7.4}$$

Here, β is a constant for the learning rate. This process is repeated a sufficiently long number of times until the policy converges.

7.3 Simulation of goal search

Let's try to implement this Actor-Critic method in a spiking network. The task is for an agent in a two-dimensional plane to search for a goal somewhere in the same plane. The original paper is Frémaux et al. (2013), and we implemented it in the LIF model. The code can be found in `code/part2/bg/` and is executed as follows.

```
node00:~/snsbook/code/part2/bg$ make
gcc -O3 -std=gnu11 -Wall -c gw.c
gcc -O3 -std=gnu11 -Wall -o gw gw.o -lm
node00:~/snsbook/code/part2/bg$ ./gw
trial = 0
:
trial = 99
node00:~/snsbook/code/part2/bg$ ls
Makefile        gw*             gw.c            gw.o            pos.dat         raster.dat
     td.dat
```

Here, `pos.dat` is the position of the agent, `raster.dat` is the spike sequence of the neuron, and `td.dat` is the data of the state value and TD error value.

The simulation results are shown in Figure 7.4. The task is to move in a straight line from the start to the goal in a two-dimensional plane with no obstacles. However, it does not know the location of the goal at first, so it takes about 50 s to find the goal after searching. After 100 trials, the agent learns to reach the goal directly for less than 5 s. (Figure 7.4A). The trajectory in the n-th trial ($n \geq 0$) can be displayed as follows.

```
node00:~/snsbook/code/part2/bg$ gnuplot
:
gnuplot> set size square
gnuplot> plot [-8:8] [-8:8] 'pos.dat' index n using 2:3 with lines
```

Here, the notation `index` n specifies the data for the n-th trial (Chapter A.2). `using 2:3` indicates that the numerical values of the coordinates in the x-y plane are used. Figure 7.4B shows the state values before and after learning. After learning, the values closer to the start are lower and those closer to the goal are higher. The reward was set to -1 for hitting a wall and $+100$ for reaching the goal. the values during the first 5000 ms in the 0th and 99th trials can be displayed as follows.

```
gnuplot> plot [0:5000] 'td.dat' index 0 with lines, 'td.dat' index 99 with lines
```

Figures 7.4C and D are raster plots of the neurons. In the first trial, the crit ic neurons continue to fire spikes with a constant firing frequency, but in the last trial, the overall firing frequency decreases and fires more strongly as the goal is approached. Looking at the activity of the Actor and State neurons, we can see that in the first trial the agent just wanders around the maze, but in the last trial it moves straight towards the goal. The raster plot can be displayed in the same way as shown below.

```
gnuplot> plot [0:5000] 'raster.dat' index 99 with dots
```

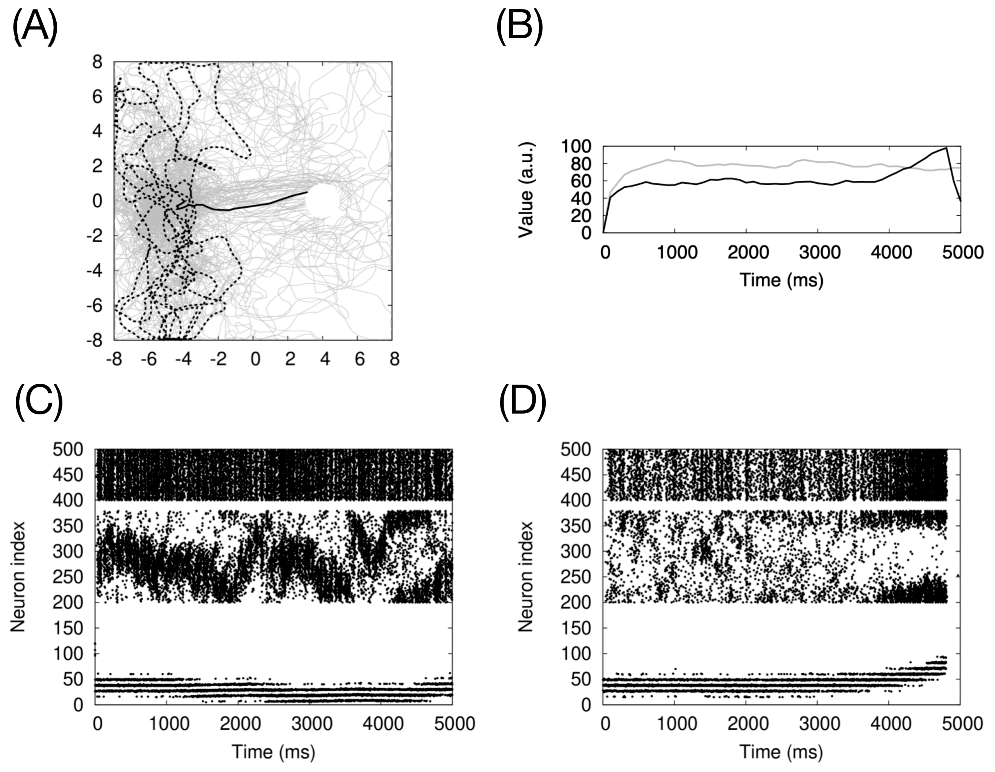


Fig. 7.4 Goal search task using the Actor-Critic method. (A) Agent's trajectories. The world is a 1.6 m $\times$ 1.6 m plane, and the start position and goal position are located respectively 40 cm left and 40 cm right, respectively, from the center. 100 trials were repeated. The trajectories of the agent in each trial are plotted in gray, and the trajectories in the first and last trials are plotted in black dashed and solid lines, respectively. (B) The state values in the first and last trials. Each is plotted in gray and black. The horizontal axis is time (ms) and the vertical axis is the value of the state value. (C) Raster plot of neurons in the first trial. Neuron numbers 0–100 represent the firings of state neurons representing the positions in the maze, 200–384 represent the firings of Actor neurons, and 400–500 represent the firings of Critic neurons. Although only the first 5 s are shown, it actually takes 50 s to reach the goal. (D) Raster plot of the neurons in the last trial. The display method is the same as in (D).

Chapter 8

Hippocampus — Simulation of associative memory

Memory and learning are important functions of the brain, and the hippocampus plays an important role in memory acquisition and recall. As a model of the hippocampus, we implement a spiking network for associative memory.

8.1 Structure and function of the hippocampus

Like the basal ganglia, the **hippocampus** is located deep in the cerebral hemisphere. The hippocampus receives input from various areas of the cerebral cortex via the **entorhinal cortex** and outputs to the cerebral cortex via the entorhinal cortex as well (Figure 8.1). There are two pathways from the entorhinal cortex to the hippocampus: the direct pathway, which provides direct input to the pyramidal cells in the **CA1** region, and the indirect pathway, which provides indirect input to the CA1 region via the granule cells in the **dentate gyrus**, which is the entrance to the hippocampus, the mossy fibers, which are the axons of the granule cells, the pyramidal cells in the **CA3** region, which receive input from the mossy fibers, and the **Schaffer collaterals**, which are the axons of the pyramidal cells. Neurons in CA1 are responsible for the final hippocampal output. The CA1 neurons are responsible for the final hippocampal output. pyramidal cells in the CA3 region are interconnected, and spikes generated in CA3 propagate to neurons in CA3 (recurrent network).

8.1.1 Types of memory

The hippocampus is known to play an important role in memory formation. Memories can be broadly classified into **declarative memory** (or **explicit memory**) and **nondeclarative memory** (or **implicit memory**). Declarative memory is further classified into **semantic memory** and **episodic memory**. Memory refers to the memory of facts, while episodic memory refers to the memory of events. Declarative memory is the memory that rises to consciousness. On the other hand, nondeclarative memories are memories that do not rise to consciousness, such as motor memories. For example, it takes a long time for a child to learn how to swim, but once he learns to swim, his body moves on its own, and he cannot explain why he learned to swim. This kind of memory for skills and habits is also called **procedural memory**.

The hippocampus plays an important role in the acquisition of declarative memories, especially short-term memories, and damage to the hippocampus prevents the acquisition of new memories. The hippocampus plays an important role in the acquisition of declarative memories, especially short-term memories, and damage to the hippocampus results in the loss of new memories. On the other hand, the hippocampus does not retain long-term memories, but is thought to play a role in converting short-term memories into long-term memories, since old memories are maintained. The

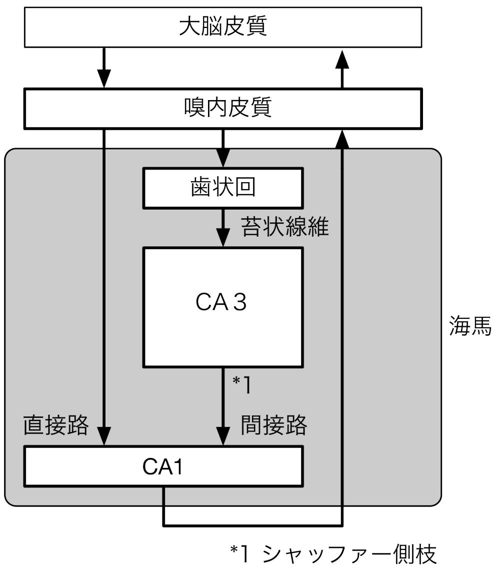


Fig. 8.1 Network structure of the hippocampus.

basal ganglia and cerebellum, as we have already seen, play an important role in the acquisition of nondeclarative memories.

8.1.2 Associative memory

One model of memory recall is the **associative memory** model. The associative memory model is a recurrent network of N neurons that embeds the pattern to be memorized into the strength of synaptic connections (Figure 8.2A). If we input a pattern that is partially different from the memorized pattern, or a pattern is missing, the rest of the pattern is complemented and the whole pattern is recalled (Nakano, 1972; Hopfield, 1982). With this model, for example, denoising is possible.

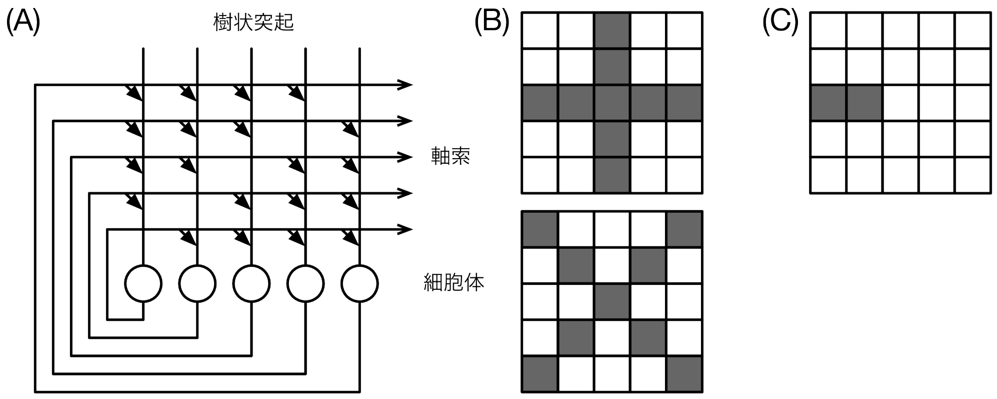


Fig. 8.2 Associative memory model. (A) Network structure. The axon of a neuron (marked with a circle) synaptically connects to the dendrites of all neurons recursively (except for those that connect to themselves). The triangular arrows represent the synapses. (B) An example of a pattern to be memorized. The top is a + pattern and the bottom is a × pattern. Black and white correspond to 0 and 1. (C) An example of a missing pattern. Only the left half of the + pattern remains.

Let $\xi^{(1)}, \ldots, \xi^{(K)}$ be the pattern we want to fill, where each pattern is a N vector of dimensions equal to the number of neurons $\xi^{(k)} = \left(\xi_1^{(k)}, \ldots, \xi_N^{(k)}\right)$. The value of each component of the vector can generally be a real number.

The synaptic coupling strength wij between neurons i and j is determined by the following equation [*1].

$$w_{ij} = \frac{1}{K} \sum_{k=1}^{K} \xi_i^{(k)} \xi_j^{(k)} \tag{8.1}$$

Then, when you give each neuron some kind of input, it will recall the pattern that is closest to that input among the embedded patterns.

As shown in Figure 8.2A, the associative memory model is a typical recurrent network. Theoretical studies have been conducted to investigate the mechanisms of memory recall and completion by viewing this as a network in the hippocampal CA3 region.

8.2 Simulation of associative memory

Let's try a simulation of associative memory, where $N = 25$ neurons are lined up in a 5×5 two-dimensional plane. Let's assume that the neurons are all recurrently connected, except for themselves, as shown in Figure 8.2A. There are two spatial patterns to be stored: a $+$ pattern and a $\times$ pattern of 5×5 pixels, as shown in Figure 8.2B. The values are set to 0 or 1, and the coupling strength between neurons is set according to Equation (8.1). The input of a pattern is done by applying an external current to the corresponding neuron for 1 s. As a missing pattern, we use a pattern with only a part of the $+$ pattern (Figure 8.2C). We have prepared this code in `code/part2/hopfield`.

8.2.1 Simulation results

First, we fed the entire $+$ pattern to the network. The firing frequency of the neurons during 1 s is plotted in Figure 8.3A, and the $+$ pattern appears depending on the firing frequency of the neurons, i.e. the memory of the $+$ pattern is recalled. On the other hand, the pattern of $\times$ is also faintly recalled, which is due to the correlation (**crosstalk**) between the patterns. Specifically, because the central pixel is common, the activity propagates to other neurons through the central neuron corresponding to this pixel. Similarly, if we give a pattern of x, the master-slave relationship is reversed. A specific population of neurons that are strongly coupled to each other and represent a specific input pattern is called a **cell assembly**.

Next, let's give a pattern in which some parts are missing. When the pattern in Figure 8.2C is given, the firing frequency of the neurons is as shown in Figure 8.3B. It is obvious that the firing frequency of the two neurons with non-zero input is high, but the other neurons corresponding to the $+$ pattern also fire with moderate intensity, indicating that the entire $+$ pattern was complemented and recalled from the missing pattern. This is accomplished by recurrent coupling of each other in the cell assembly. The execution of the code is done as follows.

```
node00:~/snsbook/code/part2/hopfield$ make
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -DSFMT_MEXP=19937 -c hopfield.c
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -DSFMT_MEXP=19937 -c ../misc/SFMT-src-1.5.1/
     SFMT.c
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -DSFMT_MEXP=19937 -o hopfield hopfield.o SFMT.o
     -lm
node00:~/snsbook/code/part2/hopfield$ ./hopfield
0 0 3
0 1 0
```

[*1] In this way, the use of products of the same pattern is called **auto-associative**.

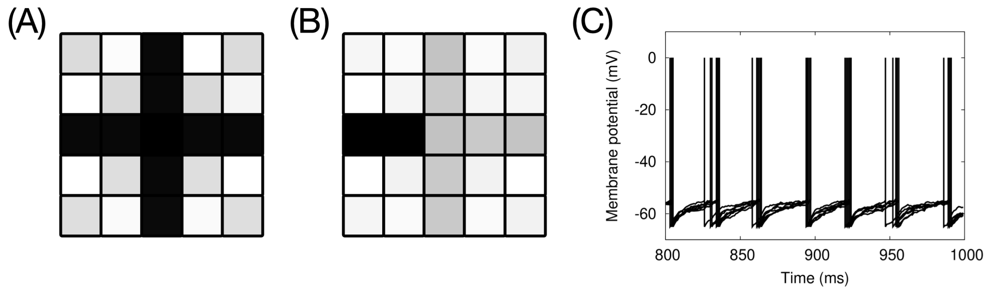


Fig. 8.3 Simulation of associative memory. (A) Firing frequency of a neuron when given a complete pattern. The firing frequency of black, gray, and white neurons averaged 158, 28, and 0 spikes/s, respectively. (B) Firing frequency of neurons when given a missing pattern. The firing frequency of black, gray, light gray, and The firing frequency of white neurons averaged 125, 25, 3, and 0 spikes/s, respectively. (C) Firing of neurons corresponding to input patterns. Pattern. The membrane potentials of the remaining seven neurons, excluding the two high-frequency firing neurons, were plotted for the last 200 ms.

```
0 2 25
0 3 0
0 4 3
:
```

Starting from the top, it shows the row number, the column number, and the firing frequency (spikes/s) of the neuron at that position [*2]. For further interest, let's look at the specific timing of spike firing instead of the firing frequency. The value of the membrane potential of neuron n ($n \geq 0$) is output to n`.dat`. If we plot the membrane potential waveforms for all but the two neurons with the highest firing frequency, we can see that the neurons corresponding to the + pattern are firing spikes at approximately the same time and in a periodically synchronized manner (Figure 8.3C). The phenomenon seen in this simulation is similar to that seen in the real brain, where neurons in a cell assembly represent a particular pattern by **synchronous firing**. These characteristics are unique to spiking neurons that explicitly handle spikes, and can only be observed by implementing them as spiking networks.

[*2]This is the only part that gnuplot could not make a good diagram for.

Chapter 9

Brain–body simulation

Since this book is about spiking network simulation, it basically introduces only simulations limited to the brain, but in studying the brain, is the brain enough? Is it enough to study the brain or is the body unnecessary? There has been a lot of discussion about whether the brain alone is enough or whether the body is unnecessary. It is true that the brain interacts with the environment through the body and cannot exist on its own. I will not discuss the rightness or wrongness of this here, but instead, let's try a simulation that includes the body.

9.1 Central pattern generator (CPG) model

Most of our typical locomotion, such as walking, is rhythmic, involving periodic repetition of certain movements. The **central pattern generator** (**CPG**) is the mechanism that generates such rhythms, and was first discovered in the lamprey lamprey. CPGs in the spinal cord control the forward swimming motion of lamprey lampreys by wriggling their bodies from side to side (Grillner et al., 1991).

Let's try to construct such rhythmic neuronal activity in spiking neurons. The Matsuoka oscillator (Matsuoka, 1985) is often used as a model for CPGs, which basically consists of two or more spontaneously firing neurons that are inhibitively coupled to each other (Figure 9.1). However, if this is the only way, both neurons may fire continuously (Section 3.5), or only one of the neurons may continue to fire, so a firing adaptation term is added (Section 3.1.4) to gradually decrease the firing frequency. In this way, the firing neurons can be replaced. In this document, we will implement the LIF model with firing frequency adaptation.

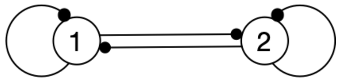


Fig. 9.1 A network of Matsuoka oscillators consisting of two neurons. The black circles arrows are inhibitory connections, and the inhibitory connections coming back to themselves represent firing adaptations.

It is essentially the same as the simulation of a network of two neurons implemented in Part I (Section 3.5.2). To this, we add the current of firing frequency adaptation. First, we create a neuron that spontaneously fires spikes by an external current and then gradually decreases its firing frequency while the neurons are not connected to each other (Figure 9.2A).

We need to connect the two inhibitory synapses. The code is `code/part2/cpg/cpg.c`. In this code, two neurons are implemented from the beginning and are not connected (the value of variable W is set to 0), so when we connect them and run the code, we get the result shown in Figure 9.2B. To run the code and check the results, you can follow the steps below.

```
node00:~/snsbook/code/part2/cpg$ make
```

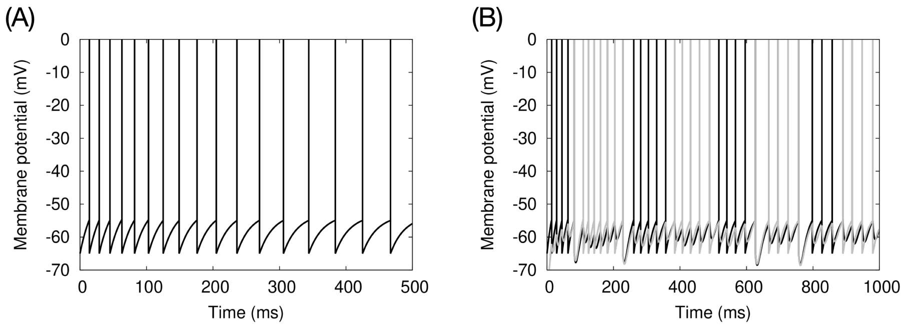


Fig. 9.2 LIF モデルを用いた松岡振動子。(A) 発火頻度の適応を加えた LIF モデルの発火パターン。(B) 2 ニューロンからなる CPG。それぞれのニューロンの膜電位を黒とグレーで表示している。$W = -2.0$ とした。横軸は時間 (ms)、縦軸は膜電位 (mV)。

```
gcc -O3 -std=gnu11 -Wall -c cpg.c
gcc -O3 -std=gnu11 -Wall -o cpg cpg.o -lm
node00:~/snsbook/code/part2/cpg$ ./cpg > cpg.dat
node00:~/snsbook/code/part2/cpg$ gnuplot
:
gnuplot> plot 'cpg.dat' using 1:2 with lines, 'cpg.dat' using 1:3 with lines
```

When one neuron is active, the other neuron is inhibited and cannot fire spikes, resulting in confirmation that the two neurons fire in alternating bursts (Matsuoka, 1985). It is important to note that the initial values of the membrane potential of the two neurons should be slightly different. Otherwise, they will start from exactly the same state, and alternating activity patterns cannot be obtained.

Similarly, three or more neurons can be made to fire in sequence by mutually connecting them with each other in an inhibitory fashion. See Matsuoka (1985) for the specific connection.

9.2 Simulation of bipedal walking with a lower limb musculoskeletal model

As a subject for the simulation of a body model using the Matsuoka oscillator, let's reimplement Taga et al. (1991) and try bipedal walking with a lower limb musculoskeletal model. This is a musculoskeletal model with five links and six joints corresponding to the lower body of a human, and the motion pattern is generated by applying torque to each joint (Figure 9.3A). The torque is provided by the CPG, which consists of six Matsuoka oscillators, each of which is assigned to a joint (Figure 9.3B). In the original paper, a rate model [*1] was used as the neuron model, but here we will reimplement it using the spiking neurons we just created in Section 9.1.

The code, including the documentation, is located at `code/part2/biped/`. By following the steps described in the included README [*2] and generating a movie, you can see how the robot actually walks on level ground with two legs (Figure 9.4). By changing the parameters, the walking pattern also changes, and we can see that the left and right motions become asymmetrical and even fall over. In Taga et al. (1991), a simulation of walking on a slope was also performed, and it was confirmed that the dynamics between the oscillators allowed the robot to continue walking on a slight incline.

[*1] Unlike the spiking neuron described in this book, this model does not fire spikes explicitly, but instead inputs and outputs analog values corresponding to the firing frequency. Also called an **analog neuron**.

[*2] The procedure is complicated and is not described in this document.

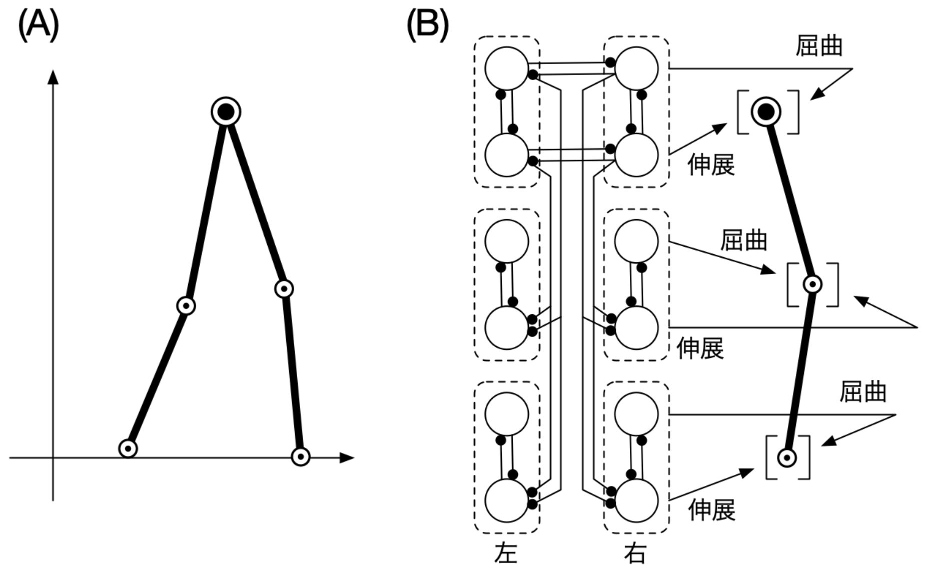


Fig. 9.3 Schematic of the reimplemented Taga et al. (1991) model. (A) Structure of the musculoskeletal system. (B) Assignment of CPGs to each joint. Each CPG is surrounded by a dashed line. Black arrows represent inhibitory connections.

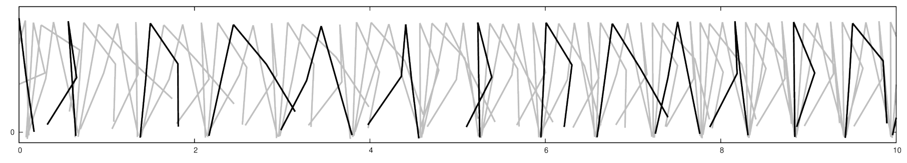


Fig. 9.4 Simulation of bipedal walking, with the robot walking for 10 meters. Black and gray are snapshots taken at 500 ms and 100 ms intervals, respectively.

Chapter 10

Self-organizing map (SOM)

We introduced the simulation of ocular dominance map formation (Section 5.1) as an example of unsupervised learning, but it was completely in the context of neuroscience. On the other hand, unsupervised learning itself is also used in engineering applications such as feature extraction and clustering. Therefore, let us try to implement Teuvo Kohonen's Self-Organizing Map (SOM) (Kohonen, 2001), a variation of unsupervised learning, in a spiking network and, in particular, let the winner be determined by the time-to-first-spike described below.

10.1 What is SOM?

The **self-organizing map** (**SOM**) is an unsupervised learning neural net, an algorithm for projecting multidimensional data onto a two-dimensional plane. It is often used for clustering and other applications, since similar data are placed close together on the plane.

The mechanism of SOM is very simple. It consists of an input layer that presents the input data (generally n-dimensional) and an output layer that projects the data onto a two-dimensional plane [*1] (Figure 10.1). In the output layer, neurons are placed in the two-dimensional plane. Every neuron receives the same **input vector** $\mathbf{v} = [v_1, \ldots, v_n] \in V$ presented in the input layer. V is a set of input data, and $\mathbf{v}$ is an n-dimensional vector. On the other hand, each neuron i has its own **reference vector** $\mathbf{w}^{(i)}$. This is also an n-dimensional vector that represents, so to speak, the strength of the synaptic connections of that neuron.

The learning procedure is as follows: Although there are many variations of SOM, we will present the simplest one here.

1. An input vector $\mathbf{v}$ is presented to the input layer.
2. The neuron with the reference vector that is closest to $\mathbf{v}$ is determined and called the **winner**. Let's say the winner is neuron i. The winner is usually calculated as the inner product of the input vector and the reference vector, and the neuron with the largest value is considered the winner. Namely,
$$i = \arg\max_i \mathbf{v} \cdot \mathbf{w}^{(i)} = \arg\max_i \sum_k v_k w_k^{(i)} \tag{10.1}$$
where $\arg\max_x f(x)$ returns the x that maximizes the value of the function $f(x)$.
3. For each neuron j in the neighborhood of neuron i, update the reference vector using the following equation.
$$\mathbf{w}^{(j)} = \mathbf{w}^{(j)} + \varepsilon \cdot h(i,j) \cdot \left(\mathbf{v} - \mathbf{w}^{(j)}\right) \tag{10.2}$$
where ε is the learning coefficient and $h(i,j)$ is the Gaussian function for the distance between

[*1] Although projection onto a multidimensional plane is also possible, 1D or 2D is generally used for ease of visualization.

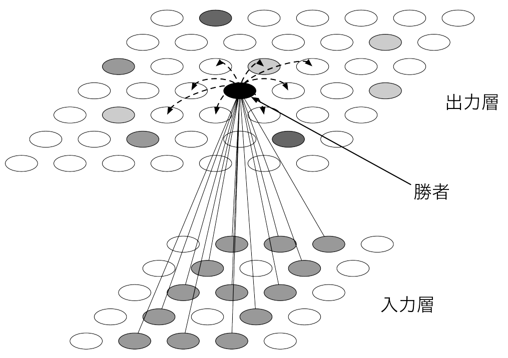


Fig. 10.1 Network structure of SOM. It is a two-layer network consisting of an input layer (bottom) and an output layer (top), each of which consists of multiple neurons. When a stimulus is presented in the input layer, the stimulus is added with a weight of synaptic connections called the reference vector, and then input to the neurons in the output layer. The most active neuron in the output layer is called the winner, and the winner updates the reference vectors of its neighbors (wavy arrows).

neurons i and j: $h(i, j) = \exp\left(-|i-j|^2/2\sigma^2\right)$. Here, $|i-j|$ is the distance between neurons i and j. As the distance increases, the value of the function converges to 0.

4. Repeat the above as many times as necessary.

Since SOM does not have the notion of completion of learning, it is necessary to gradually decrease the learning coefficient ε and gradually narrow the width of the Gaussian function σ.

10.2 Implementation of SOM and application to MNIST

Now, let's try to implement it with a spiking network. We will use **MNIST**[*2] handwritten characters as the input data. There are 60000 handwritten images of numbers 0 to 9 in grayscale of 28×28 pixels. Let's present them in order. Since the grayscale is [0, 1), we will use this value to fire Poisson spikes of up to $f_{\max} = 100$ spikes/s per pixel. The stimulus presentation period is 100 ms, and every 100 ms we move to the next image presentation. The number of neurons in the output layer is 32×32, and the reference vector is initialized using a random number in [0, 1). Equation (10.2) is used to update the reference vector [*3].

10.3 Rate coding and temporal coding

In this implementation, we will compute a winner each time we present an image of a character. Equation (10.1) corresponds to the synaptic input, so we will use it to calculate the membrane potential and fire spikes. A simple implementation would be to make the winner the neuron that fires the **most** spikes during the stimulus presentation period (100 ms). In other words, the neuron that fires the most strongly is the winner. In this way, coding that uses firing frequency to represent information is called firing frequency coding (**rate coding**). However, here we will consider coding that utilizes the characteristics of spikes. For example, let's say that the winner is the neuron that fires the **earliest** spike during the stimulus presentation period (100 ms). This means that if the intensity of the incoming external current is large, the membrane potential will reach the threshold

[*2] A large database of handwritten numeric images published by the U.S. National Institute of Standards and Technology (NIST).

[*3] Normally, we would use STDP, etc. here, but the point of this chapter is TFS, so please take it with a grain of salt.

and spike as soon as possible. This method of using the firing time of the first spike is called **time-to-first-spike** (**TFS**), and the coding that expresses information explicitly using the firing time of the spike is called **temporal coding**.

Simulations of the self-organization of the ocular dominance map (Section 5.3) showed that the map responded more strongly to inputs from either the right or left eye. The CPG simulation of bipedal walking (Section 9.2) also uses firing frequency coding, since the firing frequency of neurons directly corresponds to the torque of muscle contraction. On the other hand, the simulation of blink reflex conditioning (Section 6.3) used the on/off firing of granule cells and combined them as a population to represent the time course. This is an example of temporal coding.

10.4 Simulation results

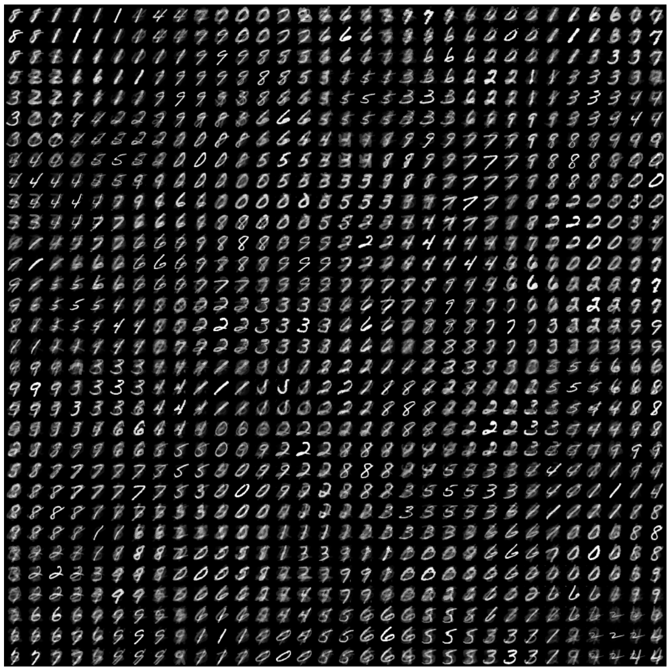


Fig. 10.2 Self-organization of a MNIST image using SOM, showing the reference vectors of 32×32 neurons in [0, 1) grayscale. Black is 0, white is 1.

I created a SOM program that determines the winner by TFS (`code/part2/som/`). you need to download the MNIST data (the instructions are in the README in the same folder), but when you run it, you get the image in Figure 10.2. This is a grayscale representation of the reference vectors of the 32×32 neurons in the output layer, so to speak, showing the strength of the synaptic connections. For example, if a neuron fires at the beginning of an input pattern, it will fire at the end of the input pattern. Since each neuron is selective for one of the letters 0–9, and nearby neurons fire strongly for similar letters, we can see how similar letters are clustered across the output layer.

This self-organization is thought to be a cortical learning algorithm, as mentioned in the simulation of ocular dominance map formation (Section 5.3). The SOM itself is mainly used for various engineering applications (Kohonen, 2001), but it is also used in neuroscience research as a model of the visual cortex (Miikkulainen et al., 2005).

Chapter 11

Approximate solutions for combinatorial optimization problems

Finally, although not directly related to the brain, I would like to introduce a method for approximating the solution of combinatorial optimization problems.

A **combinatorial optimization problem** is a problem of finding the best solution among many candidate solutions. In particular, it is a problem where the solution is expressed as a combination of several variables and the number of candidate solutions is equal to the number of variables. Certain combinatorial optimization problems have the property that it is difficult to find a solution, but easy to determine whether a given candidate is a solution or not. Because of this difficulty, it is often meaningful to find a candidate that is not exactly a solution, but is very close to a solution. Such a solution method is called an **approximate solution method**.

As a general-purpose algorithm for such approximate solutions, spiking networks are sometimes used. As an example, let's construct a spiking network for solving Sudoku.

11.1 What is Sudoku?

Sudoku is a puzzle in which all rows, columns, and blocks are filled with squares so that the numbers do not overlap (ニコリ, 2006). Usually 9×9 squares are filled with numbers 1–9. For the sake of simplicity, we will use the numbers 0–3 in the 4×4 squares. Let's consider an example shown in Figure 11.1.

2	3	0	1
3	1	2	0

2	3	0	1
1	**0**	**3**	**2**
0	**2**	**1**	**3**
3	1	2	0

Fig. 11.1 Sudoku example. (Left) Example. Fill in the blank squares so that exactly one of the numbers 0-3 appears in every row, column, and block (double-lined square). (Right) Correct answer. The numbers in bold are the solutions.

11.2 How to configure a network

Assign a neuron corresponding to each of the numbers 0-3 to each of the squares in Figure 11.1. The number of cells is $4 \times 4 = 16$, so The total number of neurons is 64.

The rules of Sudoku are represented by a pattern of connections between neurons (Figure 11.2). The first thing to consider is that each square can contain only one number from 0–3. This corre-

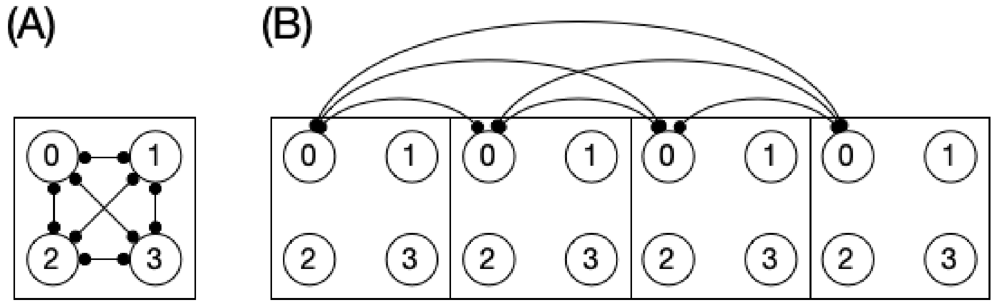


Fig. 11.2 How to create a join pattern. (A) How to make connections in a cell. Each cell contains four neurons corresponding to the numbers 0–3, and the neurons are inhibitively coupled to each other. (B) How to connect in each column. In this figure, there are four neurons corresponding to the numbers 0–3 in the same column. In this figure, the neurons corresponding to the number 0 are coupled. The circled arrows indicate inhibitory connections.

sponds to the situation where only one of the four neurons fires strongly, while the rest either do not fire or fire very infrequently. To create such a state, the four neurons in each cell should be coupled so that they inhibit each other (Figure 11.2A). In this way When a neuron fires, it suppresses the firing of neurons corresponding to other numbers, thus avoiding the simultaneous appearance of multiple numbers (i.e., neuron firing). In this way, the firing of one neuron suppresses the firing of all other neurons, which is called **winner-take-all** (**WTA**). Then, for each row, every number must appear exactly once. To represent this, for a given row, the neurons of a given number in a given square are coupled in such a way that the neurons of the same number in all the other squares of the same row are inhibited (Figure 11.2B). Each column and block should be connected in the same way. Note that all connections between neurons must be inhibitory. Excitatory connections are not necessary.

Next, let's represent the numbers that we know in advance as hints. It is easy to do this by applying a large external current to the corresponding number of neurons in the corresponding square. On the other hand, a random current is applied to all the neurons as noise. This noise current will make the activity of the neurons stochastic, and the state of the network will converge towards the solution of the Sudoku puzzle.

11.3 Simulation results

In this section, we will discuss how to make a model of a neuron and a synapse using a current-based integral firing model and an exponential decay synapse. In this case, the spike trains in the first half of the simulation are ignored and only the spike trains in the second half are used to calculate the firing frequency of each neuron.

The code can be found at `code/part2/sudoku/`. I will not explain it, but it is only 150 lines long, so I think it is decipherable enough. When you compile and run the code, you will see the solution you obtained by solving the above problem.

```
node00:~/snsbook/code/part2/sudoku$ make
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1/ -DSFMT_MEXP=19937 -c sudoku.c
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1/ -DSFMT_MEXP=19937 -c ../misc/SFMT-src-1.5.1/
    SFMT.c
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1/ -DSFMT_MEXP=19937 -o sudoku sudoku.o SFMT.o -
    lm
node00:~/snsbook/code/part2/sudoku$ ./sudoku
2 3 0 1
1 0 3 2
0 2 1 3
3 1 2 0
```

```
node00:~/snsbook/code/part2/sudoku$
```

The activity of the neurons at that time is shown in Figure 11.3. The neurons firing at a high frequency are those of the clue numbers, while the neurons in the blank squares are firing randomly.

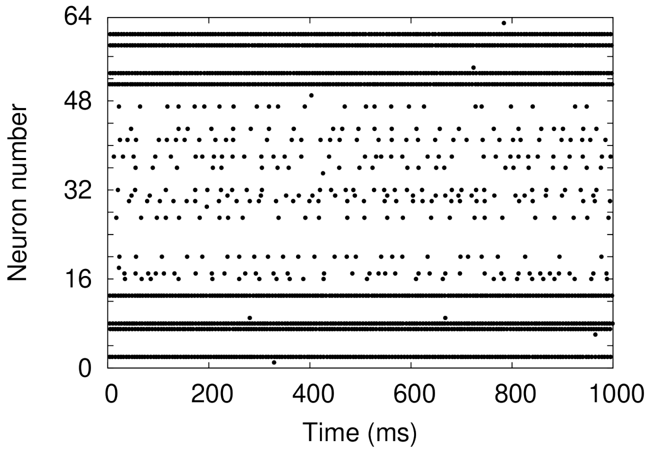


Fig. 11.3 Neuron activity while solving a Sudoku puzzle. The horizontal axis is time (ms) and the vertical axis is the neuron number. The number of neurons moves from the top left to the bottom right of the square, and increases by one in each square.

11.3.1 Notice

Note that the behavior of the network is stochastic, so it may output the wrong solution. In this sense, the network is not a deterministic algorithm, but a probabilistic and approximate one. The problem of Sudoku is defined by the program's first lines:

Listing 11.1 `sudoku.c`

```
int puzzle [ N ] [ N ] = { {  2,  3,  0,  1 },
                           { -1, -1, -1, -1 },
                           { -1, -1, -1, -1 },
                           {  3,  1,  2,  0 } };
```

where −1 is a blank square. If we rewrite this section, we can create a different problem, or create a problem with a size of 9 × 9 or even larger. We will leave that to the reader.

11.4 Column: How to use good random numbers

Random numbers are necessary for making synaptic connections, inputting Poisson spikes, and, as in the Sudoku example, adding noise currents to neurons to make them fire more erratically. The OS built-in implementation of `rand(3)` is designed to be run on old, low performance computers, so that it can be executed with a simple rule and a small amount of computation, but it may not have good properties as a random number. For example, when I run `man 3 rand` on my macOS Catarina (10.15.5), the manual says

```
RAND(3)                  BSD Library Functions Manual                  RAND(3)

NAME
     rand, rand_r, srand, sranddev -- bad random number generator

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdlib.h>
```

It is explicitly written as **bad** random number generator. So let's not use `rand(3)` anymore. The authors recommend **Mersenne Twister** (Matsumoto and Nishimura, 1998), and SFMT [*1] is a popular implementation; the latest version as of September 27, 2021 is version 1.5.1, so download it and use it. When you extract it, various files including `SFMT.c` and `SFMT.h` will be generated. The code example of how to use it is provided under `code/column/rng/`.

Listing 11.2 `main_rand.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define SEED ( 32 )

int main ( void )
{
  srand ( SEED );

  int32_t a = rand ( );
  double b = rand ( ) / ( double ) RAND_MAX;
}
```

If you are creating integers of $[0, \texttt{RAND\_MAX}^{*2}]$ or floating point numbers of $[0, 1]$ with code like that, you can rewrite as:

Listing 11.3 `main_sfmt.c`

```c
#include <stdio.h>
#include <stdint.h>
#include <SFMT.h>

#define SEED ( 32 )

extern void sfmt_init_gen_rand ( sfmt_t *, uint32_t );
extern uint32_t sfmt_rangend_uint32 ( sfmt_t * );
```

[*1] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index-jp.html`

[*2] Catalina, the definition is written in `/Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/usr/include/stdlib.h` and the value is 0x7FFFFFFF = 2147483647.

```
extern double sfmt_genrand_real1 ( sfmt_t * );

int main ( void )
{
  sfmt_t rng;
  sfmt_init_gen_rand ( &rng, SEED );

  uint32_t a = sfmt_genrand_uint32 ( &rng );
  double b = sfmt_genrand_real1 ( &rng );
}
```

and change `Makefile` to

Listing 11.4 `Makefile`

```
CC = gcc
CFLAGS = -O3 -std=gnu11 -Wall
SFMTDIR = ../misc/SFMT-src-1.5.1
SFMTFLAGS = -I$(SFMTDIR) -DSFMT_MEXP=19937

all: main_rand main_sfmt

main_rand: main_rand.o
	$(CC) $(CFLAGS) -o $@ $^

main_rand.o: main_rand.c
	$(CC) $(CFLAGS) -c $<

main_sfmt: main_sfmt.o SFMT.o
	$(CC) $(CFLAGS) -o $@ $^

main_sfmt.o: main_sfmt.c $(SFMTDIR)/SFMT.h
	$(CC) $(CFLAGS) $(SFMTFLAGS) -c $<

SFMT.o: $(SFMTDIR)/SFMT.c $(SFMTDIR)/SFMT.h
	$(CC) $(CFLAGS) $(SFMTFLAGS) -c $<

clean:
	rm -f main_rand main_sfmt *.o

distclean: clean
```

You can prepare and `make` it as follows. See also the column "How to write `Makefile`".

If you want $[0, 1)$, you can use `sfmt_genrand_real2`, and if you want $(0, 1)$, you can use `sfmt_genrand_real3`.

The random numbers directly affect the results of the simulation, so they should be used with care. Interesting results were obtained. But when I looked into it, I found that the random numbers were statistically skewed. If this happens, you are out of luck. In particular, when you are doing statistical tests, or annealing or Bayesian calculations using spiking neurons, this is an essential problem, so you should use a program whose statistical properties are properly checked and guaranteed. The Mersenne twistor, for example, is used by default in Python and Ruby, and is the first random number to try. If you are using a GPU and need faster execution, you can trade off the performance and consider other faster random numbers.

Part III

Using Supercomputers Efficiently

We have been running various neural circuit simulations throughout Parts I and II, and the number of neurons and synapses is at most several thousand and several hundred thousand, respectively, which is sufficient to run on a single ordinary PC. For the purpose of learning the basics of spiking network simulation, it makes sense to try such a small-scale model. On the other hand, the actual human brain consists of about 86 billion neurons and and trillion synapses, a simulation of this scale on a single computer is virtually impossible. This is not possible because it does not finish in time and it cannot be stored in memory.

This is where **supercomputers** come in. A supercomputer is a computer of the highest performance class of its time, and modern supercomputers are composed of a huge number of computers connected by a network. For example, the supercomputer "Fugaku" at the RIKEN Center for Computational Science (R-CCS) consists of about 160,000 computers [*3]. In addition, supercomputers equipped with a large number of special hardware for parallel computing, such as graphics processing units (GPUs), are becoming widely used. This huge number of computers is used to divide up the neural circuitry and perform calculations in parallel to speed up the computation and allow it to be loaded into memory.

The field of studying how to use supercomputers is called **high-performance computing** (**HPC**), which is already an established research field and supports many scientific and technological calculations. HPC is already a well-established research field and supports many scientific and engineering calculations. The adoption of HPC technology in the field of neuroscience is rapidly spreading, and it is essential for future neural circuit simulations (Yamazaki et al., 2021).

Therefore, in Part III, we will deal with speeding up the simulation of spiking networks by parallel computing, which we call **high-performance neurocomputing**. We will parallelize the computation using various architectures and methods, and see how fast the computation can be.

[*3] About the supercomputer "Fugaku" `https://www.r-ccs.riken.jp/jp/fugaku`

Chapter 12

Introduction to high-performance neurocomputing

Once again, a supercomputer is a computer of the highest performance class of its time. As of September 2021, Japan's supercomputer "Fugaku" boasts the world's highest performance, consisting of about 160,000 computers (compute nodes).

12.1 Performance measure of supercomputers

The computational performance of a supercomputer is evaluated in terms of **FLOPS** (Floating-point Operat ion Per Second). This value indicates how many floating-point operations can be performed in 1 second. This is the number of floating-point operations that can be performed in one second. Since modern scientific and technological calculations are mainly based on floating point calculations, FLOPS is an important indicator. For example, the performance of "Fugaku" is about 442 P (Peta) FLOPS, and since Peta represents 10^{15}, it can perform a tremendous number of basic operations, about 44 times per second. The performance of CRAY-I (1976), an early supercomputer, was about 160 M (Mega) FLOPS, which is equivalent to that of an iPhone 4S. There is a supercomputer performance contest called Top 500 [*1], which is updated twice a year (June and November), and the Top 500 uses a benchmark called High-Performance Linpack (HPL) to measure FLOPS. The first champion was CM-5 of Thinking Machines (about 60 Giga FLOPS).

The performance of supercomputers has continued to improve almost in accordance with **Moore's Law**. Moore's Law refers to the trend that the number of transistors in an integrated circuit doubles approximately every two years. The increase in the number of transistors allows for an increase in the number of arithmetic units. Therefore, with the exponential increase in the number of transistors per integrated circuit, the theoretical computing performance of integrated circuits is also improving exponentially. In fact, the Top 500 results show that this trend has been maintained for the past 30 years (Figure 12.1).

The value **B/F** (Bytes per FLOP) is also important. This is the data transfer rate (Bytes/s) divided by the operation speed (FLOPS), and indicates how many bytes can be read or written per operation. It shows how many bytes can be read or written per operation. For example, let's consider the following code:

```
double a [ N ], b [ N ], c [ N ];
:
for ( int32_t i = 0; i < N; i++ ) {
  a [ i ] = b [ i ] + c [ i ];
}
```

[*1] https://top500.org/

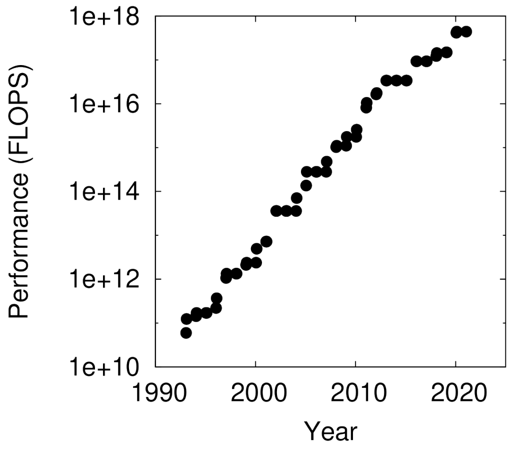


Fig. 12.1 Performance trend of the top 500 supercomputers. The horizontal axis is the year and the vertical axis is the logarithm of the performance (FLOPS). Data obtained from Wikimedia Commons.`https://commons.wikimedia.org/wiki/File:Supercomputers-history.svg`

It adds the elements of arrays `b` and `c` and stores them in array `a`. In this code, two variables of `double` are read and one is written for each loop, so 24 bytes are read and written, and the operation is one addition. Therefore, in order to execute this code at maximum speed, B/F=24 is sufficient. For example, the Fujitsu A64FX, the heart of Fugaku, has an arithmetic performance of 2.7 T (Tera) FLOPS and a memory bandwidth of 1.0 TBytes/s. So, B/F = 1.0/2.7 = 0.37. NVIDIA's latest GPU, the A100, has 9.7 TFLOPS and 1.6 TBytes/s, so B/F = 0.16. This is a surprisingly low value. In modern CPU design, B/F $\approx$ 0.5 at best, and the low memory bandwidth prevents the performance from rising to the theoretical value [*2] (Hennessy and Patterson, 2017). Currently, this problem is being overcome by the introduction of **cache memory**. Cache memory is a small but very fast memory on the CPU, and if it is used well, a much higher B/F can be obtained. For example, in the case of the A64FX, B/F = 4 if it is in the L1 cache closest to the arithmetic unit. In other words, depending on the target of the calculation, it is necessary to devise an algorithm, divide the problem to increase the efficiency of cache usage, or switch the order of calculations.

The reason for using a supercomputer is to solve large problems quickly, and the absolute performance, which is how fast the computation of a single problem is completed, is first important, but the **scaling** performance, which is how much the computation time changes when the size of the problem is changed, is also important. If the scaling is good, it means that larger problems can be solved by increasing the computational scale of the supercomputer. There are two scales of scaling: **strong scaling** and **weak scaling**. Strong scaling measures whether the computation becomes N times faster if the computational performance is increased by a factor of N for a problem of a certain size. Weak scaling measures whether the computation time will remain constant if the computational performance and problem size are increased by N times simultaneously. Weak scaling measures whether the computation time will remain constant if the computational performance and problem size are increased by a factor of N.

[*2] Called **memory wall**

12.2 What is high-performance computing?

12.2.1 Parallelization of computation

I mentioned at the beginning of this article that the field of research on how to use supercomputers is called high performance computing (HPC). Why do we need supercomputers? There are two reasons: one is that the computation is too slow to be completed on an ordinary PC CPU, and the other is that the problem is too large to fit in the memory of an ordinary PC. In both cases, the problem is divided into several smaller problems, each of which is computed on a separate computation node. In other words, the problem is divided and solved in parallel. The development and refinement of parallel computing techniques is a major research theme in HPC.

In a program, there are parts that can be parallelized and parts that cannot. If the ratio of the latter is large, no matter how much effort is put into parallelization, no performance improvement can be expected. In general, if p percent of the program is parallelizable and the performance improvement rate is s, the overall performance improvement rate by parallelization is $1/\left((1-p)+(p/s)\right)$. This is called **Amdahl's law**. Therefore, in parallelization, for example, a profiler is used to investigate where the bottlenecks in the computation are, and to determine whether parallelization will improve the overall performance. You may want to consider how much you can contribute to improving the performance of the system.

There are two main methods of parallelization: **model parallelism** and **data parallelism**. Model parallelism is a method of dividing a problem (= model) and solving it in parallel. For example, it is equivalent to solving a large neural net by dividing it. In model parallelism, information must be exchanged between the partitioned models, so communication occurs between computation nodes. If the communication time becomes too long, communication latency occurs, and the performance degrades. Thus, communication can always be a bottleneck, making it difficult to achieve performance. Data parallelism is a method of solving the same model in parallel under different parameters. For example, it is equivalent to training a neural net on different data sets. Since data parallelism is a completely independent computation, there is no need for communication between models.

12.2.2 Types of parallel computers

Computers for parallel computing can be broadly classified into two types: **shared memory** type and **distributed memory** type. In the shared memory type, multiple arithmetic units (= compute cores) share the same memory space. The CPUs of today's ordinary PCs can access the same memory on multiple cores, so this is a shared memory type. In the shared memory type, all processors can access the same information, and there is no need to duplicate data between processors, but the number of processors is limited to one. The distributed memory type is one in which multiple operators have independent memory spaces, and assumes, for example, multiple PCs connected by a network. In the distributed memory type, there is no limit to the number of computers, but each computer has different information, so if you want information from another computer, you have to communicate and copy it. Modern supercomputers are a hybrid of the shared memory type and the distributed memory type.

Libraries have been prepared to support each type of parallelization. For accelerators such as GPUs, CUDA is available for NVIDIA GPUs, and ROCm for AMD GPUs.

12.2.3 Efficiency of parallel computing

In order to execute parallel computation more efficiently, it is necessary to evaluate its efficiency. In this section, we will introduce the evaluation indices.

First, we need to know the maximum possible computational performance of the computer to be used. This is called the **theoretical performance**. This can be calculated from the specification of the computer to be used. Here, we will consider a single computer system that will be used in the examples. The computing system has 16 CPUs, each with 12 cores, each CPU core has two vector arithmetic units (Chapter 16), the length of the vector is 256 bits, and the CPU is driven at 2.2 GHz in normal mode. In the single precision case, the vector can compute 8 elements simultaneously. In the single-precision case, the vector can calculate 8 elements simultaneously, and each arithmetic unit can perform addition and multiplication simultaneously. Therefore, the number of single-precision operations that can be performed in one second is $8 \times 12 \times 2 \times 2 \times 16 \times 2.2 \times 10^9 = 13516.8$ GFLOPS. The Fugaku system consists of 158976 CPUs, each of which has 48 cores, and each CPU core has two vector arithmetic units, with a vector length of 512 bits, and the CPUs run at 2 GHz in normal mode. In the case of double precision, the vector can calculate 8 elements simultaneously. In addition, each arithmetic unit can perform addition and multiplication simultaneously. Therefore, the number of double-precision operations possible in one second for Fugaku is $158976 \times 48 \times 2 \times 2 \times 8 \times 2 \times 10^9 = 4.88 \times 10^{17} = 488$ PFLOPS [*3].

On the other hand, the total number of operations per simulation/execution time, i.e., how many operations can be performed in one second, is called **effective performance**. The total number of operations is a count of the number of operations in your program. In the case of scientific and technological calculations such as brain simulations, it is usually the number of floating-point operations that perform the numerical calculation that is counted.

Finally, the execution performance/theoretical performance is called the **execution efficiency**, which indicates how efficiently the performance of the computer was used.

It is standard practice to evaluate performance based on the maximum computational performance of the computer being used. In the authors' experience so far, in spiking network simulation, the execution efficiency rarely exceeds 50take into account memory performance and network performance, but these factors often cause performance degradation. In order to improve the performance, it is necessary to identify the causes and deal with them, and profiling is necessary, such as analyzing each part of the calculation time by taking a more detailed breakdown.

12.3 Parallelization of spiking network simulation

The simulation of the random network described in Section 3.5.4 was completed in about 17 seconds on an ordinary PC. 4,000 neurons is enough for a single PC, but if you want to run a human-scale whole-brain simulation with 86 billion neurons, a single PC will not have enough computing power or memory. This is where parallel computing comes in. This is where parallel computing comes into play.

12.3.1 Parallelization, first thing first

There are several important rules for parallelization, the first of which is the following.

Rule 1. If there is room for tuning in the code for sequential computation, do not parallelize it.

In other words, before starting parallelization, we should first examine whether there is any waste in the code of sequential computation that is not parallelized, and try to make it fast enough. However, where should we start and what should we do to speed up sequential processing? First of all, it is important to understand the part of the target calculation that is taking time. In neural circuit simulation calculations, it often takes time to iterate in a loop. For example, looping through

[*3] `https://www.r-ccs.riken.jp/fugaku/system/`

multiple neurons, or looping through time steps. Next, let's look at the repetitive processes in the loop. For example, suppose that the process includes multiplication, addition, subtraction, and division. In fact, the time required for the four arithmetic operations is not the same. Although it depends on the type of calculator, division often takes a very long time, sometimes ten times longer than other operations. In many cases, division can be rewritten as multiplication by a program. For example, 1/2 can be changed to ×0.5. Just by doing this, every time you repeat a loop, you can avoid division and increase the speed.

Parallelization should be considered only when the performance is still unsatisfactory after sufficient speedup. In such a case, use Amdahl's law to check whether there is enough room to speed up the target computation in the computing environment you are using before starting parallelization. Blind parallelization may introduce unnecessary bugs and induce unexpected behaviors that do not appear in sequential execution. For example, there is a phenomenon called **deadlock**. This is a situation in which multiple processing units executing parallel processing wait for each other to finish their processing, preventing the processing from proceeding. When this happens, the program needs to be analyzed and modified, which is not necessary in sequential processing.

12.3.2 Sparse matrix storage

Let's take the simulation of a random network implemented in Section 3.5.4 as an example of how to speed up a program. Since this simulation takes almost 20 seconds to compute, we would like to consider speeding up the simulation by parallelization, but is there any room for improvement?

There is one obvious point of improvement. The coupling matrix `w` between the neurons is an $N \times N$ matrix. Now, $N = 4000$, so it is a very large matrix. On the other hand, most of the matrix elements (98%, to be specific, since the join probability P is 0.02) are zero. Such a matrix with most of the elements being zero is called a **sparse matrix**. If we can skip the zero part, 98% of the calculation can be omitted.

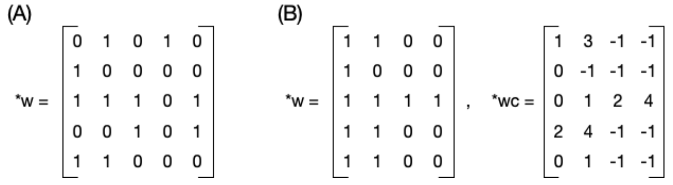


Fig. 12.2 How to store a sparse matrix in ELL format. (A) An example sparse matrix. (B) Representation in ELL.

So, let's improve the storage method of the joint matrix `w`. There are various methods for storing sparse matrices. One of the most common is **CSR** (Compressed Sparse Row, or Compressed Row Storage; **CRS**). However, in the case of spiking network simulation, the **ELL** format is more natural. In the ELL format, we have two arrays: one to store the values of the non-zero elements per row, and the other to store the column numbers of the non-zero elements per row (Figure 12.2). The number of synapses to be coupled per row (i.e. per post-side neuron) is different, so we reserve each array for the maximum number of synapses and fill in the missing values as needed. For example, the value array should be filled with 0, and the column number array should be filled with −1. The latter can also serve as a guard to indicate the end of the loop by filling it with −1. In that way, Listing 12.1 becomes the same as Listing 12.2 (all the code is in `code/part3/random/`).

Listing 12.1 `code/part3/random/random.c:calculateSynapticInputs`

```
for ( int32_t i = 0; i < N; i++ ) {
  float re = 0, ri = 0;
  for ( int32_t j = 0; j < N; j++ ) {
    float r = n -> w [ j + N * i ] * n -> s [ j ];
    if ( j < N_E ) { re += r; } else { ri += r; }
  }
:
```

Listing 12.2 `code/part3/random/random_ell.c:calculateSynapticInputs`

```
for ( int32_t i = 0; i < N; i++ ) {
  float re = 0, ri = 0;
  for ( int32_t j = 0, k = 0; ( k = n -> wc [ j + n -> nc * i ] ) != -1; j++ ) {
    float r = n -> w [ j + n -> nc * i ] * n -> s [ k ];
    if ( k < N_E ) { re += r; } else { ri += r; }
  }
:
```

where `w` is the matrix that stores the values, `wc` is the matrix that stores row numbers, and `k` is a presynaptic neuron number. Note that the value of `k` is taken from `wc`. Also, the variable `n - > nc` is the maximum number of synapses that will be coupled to a single post-side neuron.

The code rewritten in this way is prepared in `random_ell.c`. The execution of this code finishes in a little over 0.64s. This is a speedup of roughly 26 times with only a few modifications. This means that there is no waiting time, and you will feel as if the computation is finished in an instant. This brings us to the next rule.

Rule 2. If you are satisfied with the performance of sequential computation, don't parallelize it.

Of course, the above calculation takes less than 1 s because there are 4,000 neurons, but if you increase the number of neurons, the calculation will take more time. The second rule is to consider parallelization only when you have to run such a large simulation.

In the following chapters, we deliberately lengthened the computation time by multiplying the binding probability P by 10 and setting the values of ge and gi to 1/10 so as not to change the strength of the synaptic input. In this case, the computation time was 4.1 s for the ELL version.

Chapter 13

Parallelizing computation with OpenMP

Let's start with OpenMP, which can only be used on a single compute node, but is worth trying first since it can be automatically parallelized by adding a few lines (often one). We will parallelize the computation of synapses and neurons in a random network.

13.1 Random network revisited

Let's review the random network program. The code for the main loop part was as follows.

Listing 13.1 `random_ell.c:loop`

```
139  for ( int32_t nt = 0; nt < NT; nt++ ) {
140    calculateSynapticInputs ( n );
141    updateCellParameters ( n );
142    outputSpike ( nt, n );
143  }
```

The synaptic inputs were calculated by the function `calculateSynapticInputs` in line 136, and the parameters of the neurons were updated by the function `updateCellParameters` in line 137. Each of these functions is called as:

```
void function_name ( network_t *n )
{
  for ( int32_t i = 0; i < N; i++ ) {
    Calculation for postsynaptic neuron i
  }
}
```

For each post-side neuron i, we compute it sequentially.

The strategy of parallelization is to execute the contents of the loop for this neuron i in parallel.

13.2 Using OpenMP

OpenMP is a standard for the parallel use of multiple CPUs and their compute cores in a single compute node (Chandra et al., 2000). The computer architecture envisioned by OpenMP is shown in Fig. 13.1. As explained in Section 12.2.2, it has a structure in which multiple CPUs with multiple cores share memory (shared memory type). In OpenMP, only one program process is launched, and its inner instructions are executed in parallel by multiple cores. The unit of computation in this case is called a **thread**. Therefore, the parallel model of OpenMP is called shared memory type **thread parallelism**.

OpenMP is a standard feature of the GNU Compiler Collect ion (GCC) C compiler [*1]. In the

[*1] Version 4.9 and later

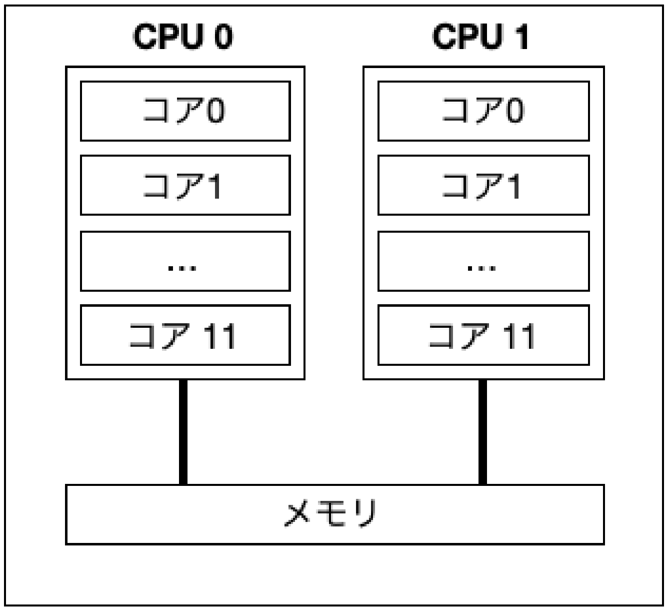


Fig. 13.1 Conceptual diagram of the computer architecture envisioned by OpenMP. Multiple CPUs with multiple cores share the memory. In particular, this figure shows an example of two CPUs with 12 cores, which is an example of the computer used by the authors.

header area of the source code, use `#include <omp.h>` and add the `-fopenmp` option at compile time, and you are ready to use OpenMP.

Parallelization of neuron loops with OpenMP is very easy. First, let's look at the function `calculateSynapticInputs`.

Listing 13.2 `random_omp.c:calculateSynapticInputs`

```
void calculateSynapticInputs ( network_t *n )
{
  #pragma omp parallel for
  for ( int32_t i = 0; i < N; i++ ) {
    float re = 0, ri = 0;
    for ( int32_t j = 0, k = 0; ( k = n -> wc [ j + n -> nc * i ] ) != -1; j++ ) {
      float r = n -> w [ j + n -> nc * i ] * n -> s [ k ];
      if ( k < N_E ) { re += r; } else { ri += r; }
    }
    n -> ge [ i ] = exp ( - DT / TAU_E ) * n -> ge [ i ] + re;
    n -> gi [ i ] = exp ( - DT / TAU_I ) * n -> gi [ i ] + ri;
  }
}
```

As you can see, all you have to do is to add one line `#pragma omp parallel for` just before the loop for variable i. With this one line, the next `for` statement will be automatically parallelized. No other modifications are necessary. However, as a prerequisite, the calculations in the loop must be completely independent for each i. In this case, the variables `re`, `ri`, `j`, `k`, and `r` are locally defined variables for each i, the arrays `n -> w` and `n -> s` are shared but read only, and the arrays `n -> ge` and `n -> gi` are accessed only by the index i, so they are all fine.

The same is true for the function `updateCellParameters`.

Listing 13.3 `random_omp.c:updateCellParameters`

```
void updateCellParameters ( network_t *n )
{
```

```
#pragma omp parallel for
for ( int32_t i = 0; i < N; i++ ) {
  n -> v [ i ] += DT * ( - ( n -> v [ i ] - V_REST ) + G_E * n -> ge [ i ] + G_I * n -> gi [ i
    ] ) / TAU_M;
  n -> s [ i ] = ( n -> v [ i ] > THETA );
  n -> v [ i ] = ( n -> s [ i ] ) * V_RESET + ( ! n -> s [ i ] ) * n -> v [ i ];
}
```

The computer assumed in this document is equipped with two 12-core CPUs, so it can run 24 threads simultaneously. Let's actually compile the program and run the calculations.

```
node01:~/snsbook/code/part3/random$ make random_omp
gcc -O3 -std=gnu11 -Wall -fopenmp -I../misc/SFMT-src-1.5.1 -D SFMT_MEXP=19937 -c random_omp.c
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -D SFMT_MEXP=19937 -o SFMT.o -c ../misc/SFMT-
    src-1.5.1/SFMT.c
gcc -O3 -std=gnu11 -Wall -o timer.o -c ../misc/timer.c
gcc -O3 -std=gnu11 -Wall -fopenmp -I../misc/SFMT-src-1.5.1 -D SFMT_MEXP=19937 -o random_omp
    random_omp.o SFMT.o timer.o -lm
node01:~/snsbook/code/part3/random$ ./random_omp
# of omp threads = 24
Elapsed time = 0.498892 sec.
```

The computation time was about 0.50 s, which is about 8 times faster than the original computation time. Theoretically, it should be 24 times faster, but due to various overheads (extra processing other than computation, such as thread control and synchronization between cores), the performance does not improve that much. However, since it is 8 times faster with just two additional lines, it should be used aggressively.

There are various other functions of OpenMP, but I will leave the explanation of them to other technical books and limit my discussion to neural circuit simulation.

Chapter 14

Parallelizing computation with MPI

In OpenMP, only a single computation node could be used, but a supercomputer usually consists of multiple computation nodes. For example, the supercomputer ”Fugaku” consists of about 160,000 computation nodes. In such a configuration, the CPU in each computation node cannot directly read or write the contents to the memory of another computation node. Therefore, it is necessary for the computation nodes to communicate with each other via the network to send and receive the contents of the memory. THis is done by **MPI** (**Message Passing Interface**). Following OpenMP, we will introduce a parallelization method using MPI.

14.1 Using MPI

MPI is a standard for parallel computing (Gropp et al., 1999), and there are a variety of implementations, ranging from open-source ones with publicly available code to proprietary ones from network equipment vendors. A schematic structure of computer network considered is shown in Figure. 14.1.

OpenMP, a single process is invoked and its internal computation is executed in parallel by threads. In MPI, on the other hand, an independent process is launched for each CPU and core, and in general, multiple processes run at once. This kind of parallelization is called **process parallelism**. Also, since OpenMP considers only a single computation node, the memory contents are shared among the CPU cores, but in MPI, the memory contents are distributed among multiple computation nodes. We have already explained that such a configuration is called a distributed memory type (Section 12.2.2). Therefore, MPI is called a distributed memory type **process parallel**.

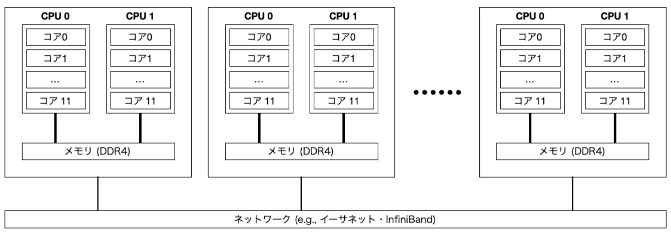


Fig. 14.1 Conceptual diagram of the computing architecture envisioned by MP I. Multiple compute nodes are connected to a high-speed network. Each compute node has multiple CPUs with multiple cores sharing memory, as in the case of OpenMP.

MPI provides various instructions such as synchronous and asynchronous send/receive, synchronization of overall control, reduction, etc. The code tends to be complex because it can be written at a very low level. In this document, we introduce parallelization using the `MPI_Allgather` instructions, which are the easiest to write.

MPI requires a little more preparation than OpenMP. First, create a file that lists the nodes to be used for the calculation.

Listing 14.1 `hostfile`

```
node01
node02
node03
node04
node05
node06
node07
node08
```

This file is called the host file, and is specified as an argument when the program is executed [*1].

The next step is to modify the program. First, the header file `mpi.h` is required.

Listing 14.2 `random_mpi.c`

```
#include <mpi.h>
```

In MPI, multiple processes are started at once, and each process is assigned a unique number called a **rank**. It is also necessary to stop all processes at the end of the computation. Such pre-processing and post-processing are described explicitly. The `main` function is as follows:

Listing 14.3 `random_mpi.c:main`

```
164 int main ( int argc, char *argv [ ] )
165 {
166   MPI_Init ( &argc, &argv );
167   int mpi_size, mpi_rank;
168   MPI_Comm_size ( MPI_COMM_WORLD, &mpi_size );
169   MPI_Comm_rank ( MPI_COMM_WORLD, &mpi_rank );
170
171   network_t n;
172
173   initialize ( mpi_size, mpi_rank, &n );
174   loop ( mpi_size, mpi_rank, &n );
175   finalize ( mpi_size, mpi_rank, &n );
176
177   MPI_Finalize ( );
178 }
```

`MPI_Init` (line 166) and `MPI_Finalize` (line 177) perform the pre-processing and post-processing, respectively. The next step is to assign ranks as

Listing 14.4 `random_mpi.c:main`

```
  int mpi_size, mpi_rank;
  MPI_Comm_size ( MPI_COMM_WORLD, &mpi_size );
  MPI_Comm_rank ( MPI_COMM_WORLD, &mpi_rank );
```

Here, the overall number of processes is stored in `mpi_size` by `MPI_Comm_size`, and the rank number of the process is stored in `mpi_rank` by `MPI_Comm_rank`. Here, `MPI_COMM_WORLD` is a label, called a **communicator**, that is shared by all the computation nodes participating in the calculation. MPI can also define a smaller group of computation nodes in the middle of the calculation, and the communicator can be used to easily distinguish the group. In MPI, it is also possible to define smaller populations of computational nodes during the course of a computation, and to easily distinguish these populations, such a communicator can be used. The resulting number of processes and rank values are then passed to functions as follows:

Listing 14.5 `random_mpi.c:main`

[*1] The MPI implementation used in this document is mvapich2 v2.1.

```
173   initialize ( mpi_size, mpi_rank, &n );
174   loop ( mpi_size, mpi_rank, &n );
175   finalize ( mpi_size, mpi_rank, &n );
```

The beginning of the function `initialize` is modified from `random_ell.c`.

Listing 14.6 `random_mpi.c:initialize`

```
43  void initialize ( const int mpi_size, const int mpi_rank, network_t *n )
44  {
45    *n = ( network_t ) { . v  = calloc ( N, sizeof ( float ) ),
46                         . ge = calloc ( N, sizeof ( float ) ),
47                         . gi = calloc ( N, sizeof ( float ) ),
48                         //. s  = calloc ( N, sizeof ( bool ), ), // defined elsewhere
49                         //. file = fopen ( "spike.dat", "w"), // defined elsewhere
50    };
51
52    // PRNG
53    sfmt_init_gen_rand ( &n -> rng, 23 );
54
55    // File
56    if ( mpi_rank == 0 ) { n -> file = fopen ( "spike.dat", "w"); }
57    :
```

First, we pass the size and rank as arguments to the function. Next, we define the structure and initialize the random numbers, and after that, we prepare the file, but modified as

Listing 14.7 `random_mpi.c:initialize`

```
55    // File
56    if ( mpi_rank == 0 ) { n -> file = fopen ( "spike.dat", "w"); }
```

In the case of MPI, multiple processes run at the same time, and if nothing is done, all of them will try to create a file with the same name. To prevent this, we will make sure that only the representative process, in this case the process with rank 0, creates the file. If we also look at the function `finalize`, we can see that:

Listing 14.8 `random_mpi.c:finalize`

```
102 void finalize ( const int mpi_size, const int mpi_rank, network_t *n )
103 {
104   :
105   if ( mpi_rank == 0 ) { fclose ( n -> file ); }
106 }
```

and modified so that only processes with rank 0 will close the file. If you forget this modification and all processes try to close the file, the program may exit abnormally. Returning to the function `initialize`, the original code defined an array `n - > s` to store the spikes, but we will remove it from here (line 48) as we will define it later to fit the MPI way.

The following function `loop` is the core of the modification of the program.

Listing 14.9 `random_mpi.c:loop`

```
142 void loop ( const int mpi_size, const int mpi_rank, network_t *n )
143 {
144   const int32_t n_each = ( N + mpi_size - 1 ) / mpi_size;
145   const int32_t n_offset = n_each * mpi_rank;
146   n -> s  = calloc ( n_each * mpi_size, sizeof ( bool ) );
147
148   bool s_local [ n_each ];
149
150   timer_start ( );
151
152   for ( int32_t nt = 0; nt < NT; nt++ ) {
153     calculateSynapticInputs ( n_each, n_offset, n );
154     updateCellParameters ( n_each, n_offset, n );
155     for ( int32_t i = n_offset; i < MIN ( n_offset + n_each, N ); i++ ) { s_local [ i - n_offset
         ] = n -> s [ i ]; }
156     MPI_Allgather ( s_local, n_each, MPI_C_BOOL, n -> s, n_each, MPI_C_BOOL, MPI_COMM_WORLD );
157     if ( mpi_rank == 0 ) { outputSpike ( nt, n ); }
158   }
159
160   double elapsedTime = timer_elapsed ( );
161   if ( mpi_rank == 0 ) { printf ( "Elapsed␣time␣=␣%f␣sec.\n", elapsedTime); }
162 }
```

First, determine the number of neurons `n_each` that each process is responsible for computing (line 144). Basically, we can divide the total number of neurons, `N`, by the total number of processes, `mpi_size`, but if we simply divide, there will be a remainder, so we need to decide a little more so that we can retain the remainder. Next, the first number of the neuron that each process is responsible for is stored in `offset` (line 145). The process with rank 0 is responsible for the first `n_each` neurons, and the process with rank 1 is responsible for the next `n_each` neurons. Since the same applies, `n_offset` is calculated as follows:

Listing 14.10 `random_mpi.c:loop`

```
145   const int32_t n_offset = n_each * mpi_rank;
```

We also define an array `n -> s` that stores the spikes in each process (line 146).

Listing 14.11 `random_mpi.c:loop`

```
146   n -> s  = calloc ( n_each * mpi_size, sizeof ( bool ) );
```

Note that the total number of neurons is not `N`, but `n_each * mpi_size`.

Next, we prepare an array `s_local` that stores the spikes of the neurons that each process is responsible for computing (line 148). The size of the array will be `n_each`. Then we enter the time step loop.

The content of the loop is as follows:

Listing 14.12 `random_mpi.c:loop`

```
153     calculateSynapticInputs ( n_each, n_offset, n );
154     updateCellParameters ( n_each, n_offset, n );
155     for ( int32_t i = n_offset; i < MIN ( n_offset + n_each, N ); i++ ) { s_local [ i - n_offset
         ] = n -> s [ i ]; }
156     MPI_Allgather ( s_local, n_each, MPI_C_BOOL, n -> s, n_each, MPI_C_BOOL, MPI_COMM_WORLD );
157     if ( mpi_rank == 0 ) { outputSpike ( nt, n ); }
```

Functions `calculateSynapticInputs` and `updateCellParameters` determine neurons to be calculated by `n_each` and `n_offset`.

Listing 14.13 `random_mpi.c:calculateSynapticInputs`

```
113 void calculateSynapticInputs ( const int32_t n_each, const int32_t n_offset, network_t *n )
114 {
115   for ( int32_t i = n_offset; i < MIN ( n_offset + n_each, N ); i++ ) {
116   :
117 }
```

Listing 14.14 `random_mpi.c:updateCellParameters`

```
127 void updateCellParameters ( const int32_t n_each, const int32_t n_offset, network_t *n )
128 {
129   for ( int32_t i = n_offset; i < MIN ( n_offset + n_each, N ); i++ ) {
  :
}
```

Each process computes only n_each neurons from n_offset. However, only the remainder of the last-ranked process is computed, so we need to devise an exit condition for the loop. The rest of the contents are the same. In this way, the number of neurons calculated by each process can be reduced from N to n_each. Next, the presence or absence of spike firing is calculated in the updateCellParameters, and it is copied to the s_local.

Listing 14.15 random_mpi.c:loop

```
    for ( int32_t i = n_offset; i < MIN ( n_offset + n_each, N ); i++ ) { s_local [ i - n_offset
    ] = n -> s [ i ]; }
```

At this stage, each process holds only the portion of the neuron responsible for the computation in terms of neuron spikes.

The following is the most important operation. Restore n -> s by exchanging the contents of s_local that each process has.

Listing 14.16 random_mpi.c:loop

```
    MPI_Allgather ( s_local, n_each, MPI_C_BOOL, n -> s, n_each, MPI_C_BOOL, MPI_COMM_WORLD );
```

This one line will do the desired exchange (Figure 14.2). After this call, array n -> s stores spikes of all neurons, so we cal start calculation for the next time step. Here, MPI_C_BOOL is the type of array element to be exchanged, and at the end of the loop, only the process with rank 0 will write data to the file.

Listing 14.17 random_mpi.c:loop

```
    if ( mpi_rank == 0 ) { outputSpike ( nt, n ); }
```

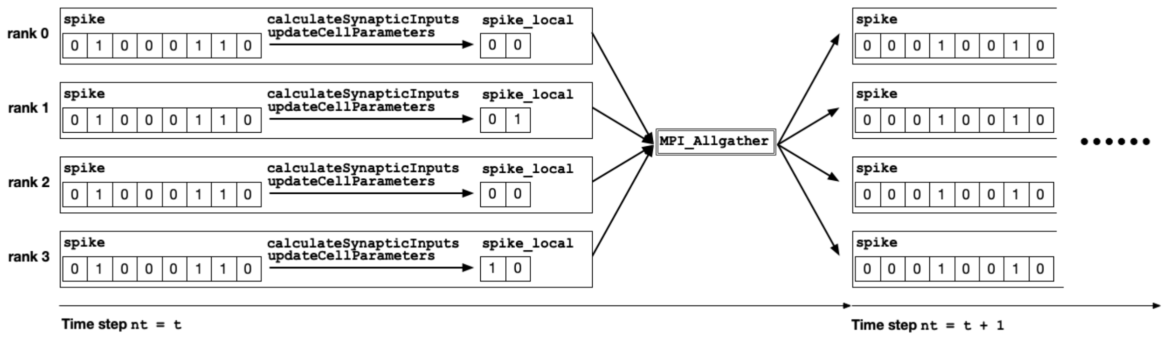


Fig. 14.2 An example of MPI_Allgather operation, with 8 neurons and 4 processes. At time step nt = t, the array spike is kept the same for each process. Each process computes only 8/4 = 2 neurons and stores the spike information in the array spike_local. Then, when MPI_Allgather is run, the spike_local is exchanged and the contents of spike are filled. Then the next step of the calculation begins.

The code using MPI is compiled with mpicc and executed with mpirun as follows.

```
node00:~/snsbook/code/part3/random$ make random_mpi
mpicc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -D SFMT_MEXP=19937 -c random_mpi.c
```

```
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -D SFMT_MEXP=19937 -o SFMT.o -c ../misc/SFMT-
    src-1.5.1/SFMT.c
gcc -O3 -std=gnu11 -Wall -o timer.o -c ../misc/timer.c
mpicc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1 -D SFMT_MEXP=19937 -o random_mpi random_mpi.o
     SFMT.o timer.o -lm
node00:~/snsbook/code/part3/random$ mpirun -hostfile hostfile -np 192 ./random_mpi
Elapsed time = 0.093905 sec.
```

The argument of `-np` is the number of CPU cores actually used for the calculation. There are 192 cores in total, so when I used all of them, the calculation was completed in about 0.1s. The sequential version took 4.1 s, which means it is more than 40 times faster. This is with only a few, almost obvious, modifications to the program.

14.2 Scaling performance

Now that we know that it is 40 times faster in absolute performance, let's look at scaling.

14.2.1 Strong scaling

Let's find out the computation time when the number of processes is increased by a factor of 2: $1, 2, 4, 8, \cdots$. The results are shown in Figure 14.3. As a whole, doubling the number of processes

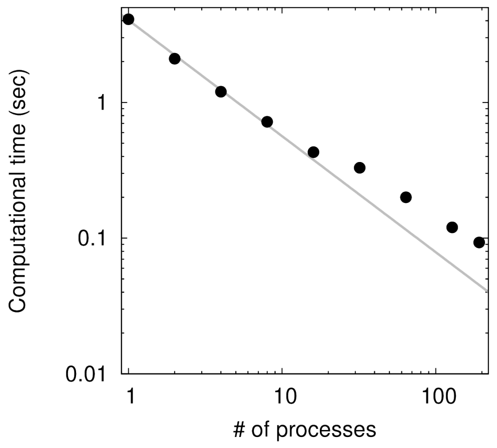


Fig. 14.3 Strong scaling performance. Computation time for each simulation when the number of processes is varied from 1 to 192. The horizontal and vertical axes are both logarithmic. The dots are the measured values, and the solid line is fitted with the exponential function $f(x) = a \cdot x^b$ The values of a and b are 4.0 and -0.85, respectively.

almost halves the computation time, so good strong scaling is obtained. On the other hand, as the number of processes increases, the computation time does not decrease so much. This is thought to be due to the overhead caused by the increase in the number of processes and also due to the increase in inter-process communication. As shown in Amdahl's law, when the number of parallel processes is increased, the ratio of the portion that can be parallelized (computation) to the portion that cannot be parallelized (communication) becomes smaller. Therefore, as the number of parallels increases, the effect of parallelization becomes smaller.

14.2.2 Weak scaling

The next step is weak scaling. In this case, the number of synapses is proportional to the number of processes. In the current code, each neuron receives synaptic connections from other neurons with probability P. Therefore, when N is increased, the total number of synapses also increases in proportion to N^2. However, the number of synapses that a single neuron can have is also increased. However, the number of synaptic connections that a neuron can have is not biologically or physically infinitely large. The number of synapses in each neuron is fixed at an average of $4000P = 800$, and only the number of neurons is varied. This assumption is based on the recent development of supercomputers. It has also been used in benchmarking spiking networks (Helias et al., 2012).

The calculation results are shown in Figure 14.4. As the computation scale increases, the computation time gradually increases, but it does not change dramatically. This indicates that relatively

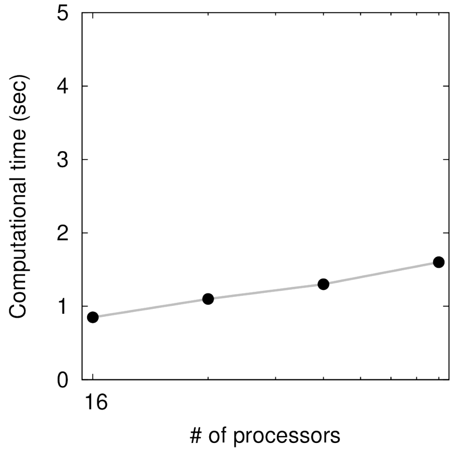


Fig. 14.4 Weak scaling performance. The number of processes was increased from 8 to 64 by a factor of 2, and at the same time the number of neurons was increased by a factor of 2.

good weak scaling is obtained. In weak scaling, the scale of computation per computation node is constant, and the amount of computation per computation node is also constant. However, the overhead for communication and synchronization between computation nodes may increase as the number of computation nodes increases, resulting in an increase in computation time as seen in the results.

14.3 Flat MPI and hybrid parallelization

The above parallelization, in which MPI communication is performed for the entire system without distinction between intra-node and extra-node, is called **flat MPI**. In the case of flat MPI, the fastest time of 0.1 s was obtained by using 192 processes. On the other hand, since the memory contents are shared among the nodes, MPI communication is essentially unnecessary, and the extra load on the network is thought to be caused by this. This can be ignored for clusters with as few as 8 nodes used in this book, but for modern supercomputers, such as Fugaku, which has 160,000 nodes, each node has 48 cores, so the number of processes can reach 7.68 million. In addition, the memory used for MPI becomes too large, leading to a serious decrease in the amount of memory available for computation. Therefore, for large clusters, **hybrid parallelism** is effective, using OpenMP inside the nodes and MPI outside the nodes, and is recommended for supercomputers such as Fugaku.

If you want to try hybrid parallelism, use `random_mpi.c` as the base, include the header file `omp.h` and add OpenMP `#pragma` statements to the function `calculateSynapticInputs` and `updateCellParameters`. Then, specify the number of nodes as `-np 16` when executing `mpirun`. At this time, we set

```
#pragma omp parallel for num_threads ( 11 )
```

to specify the number of threads as `num_threads ( 11 )`. The reason for not using all threads is that one is used for the MPI process. If the program is `random_hyb.c`, then

```
node00:~/snsbook/code/part3/random$ mpirun -hostfile hostfile -np 16 ./random_hyb
Elapsed time = 0.296803 sec.
```

This is not as fast as the fastest flat MPI, but it is still as fast as 32 parallel.

Since hybrid parallelism incurs the overhead of two parallel methods, in this case MPI and OpenMP, the computational performance may be lower than that of flat MPI. However, as mentioned earlier, in a parallel computing environment that exceeds tens of thousands of computers, it is necessary to reduce the communication load and memory consumption, and this is where hybrid parallelism becomes advantageous.

14.4 Performance evaluation

Finally, let's evaluate the execution efficiency.

In the case of the example simulation of a random network of 1 s (1000 steps), the computation time is dominated by the large number of synaptic connections. The calculation of synaptic conductance occurs $4000 \times 0.2 \times 2 \times 4000 \times 1000 = 6.4 \times 10^9$ times, since 4000 cells have a 20% chance of connecting to another neuron. On the other hand, the calculation of the membrane potential is $11 \times 4000 \times 1000 = 44 \times 10^6$ times. This is very small and can be ignored. Therefore, the total number of operations is about 6.4 Giga cycles. Since this operation was completed in about 0.1 s, the execution performance is about 6.4 Giga cycles / 0.1 s = 64 GFLOPS. The execution efficiency can be calculated as 64 GFLOPS / 13516.8 GFLOPS = 0.0058, or 0.58%.

Chapter 15

Parallelizing computation with GPU

Graphics Processing Units (**GPUs**) were originally developed for parallel processing of pixel brightness, objects, and other image processing in games and computer graphics. Since the mid-2000s, there has been a movement to use its parallel computing performance for scientific and technological calculations, and the development of computing environments has progressed rapidly. From the mid-2000s, the movement to use the parallel computing performance for scientific and technological calculations gained momentum, and the development of computing environments rapidly progressed. One of the most representative examples is the CUDA (Compute Unified Device Architecture) developed by NVIDIA, which allows users to write programs using APIs for parallel computing.

In this section, we will use NVIDIA Tesla V100 as the GPU and CUDA as the computing environment to perform parallel computation of random networks.

15.1 What is a GPU?

First, let's look at the structure of the GPU: the NVIDIA V100 has 5120 CUDA cores, which are specialized units for computing, and runs at 1.53 GHz. The number of cores in a typical CPU is only a few dozen, so the number of cores in a GPU is more than 100 times as many, even though they are specialized for computing.

CUDA cores are arranged in units called Streaming Multiprocessors (SMs), which have a hierarchical structure. The V100 has 80 SMs (Figure 15.1). Each SM has 64 single-precision CUDA cores, 64 integer processor cores, 128KB of L1/shared memory, and 256KB of register files. Furthermore, 32 double-precision CUDA cores and 8 Tensor cores for matrix computation are installed. L1/shared memory is a small, high-speed memory that can be allocated between an area to act as L1 cache memory and a user-specified **shared memory** area (Section 15.3).

Next, let's look at CUDA's programming model, which is tailored to the hierarchical structure of GPUs and consists of a hierarchical structure of computational units: **grids**, **blocks**, and threads (Figure 15.1). Threads are assigned to CUDA cores and are the entities that perform processing. A collection of threads is a block, and a block is assigned to an SM. A collection of blocks is a grid, which is the entire processing assigned to the GPU.

To get the most out of the V100 in this programming model, one might think that a grid with 80 blocks of 64 threads would do the trick. However, to make efficient use of the CUDA cores, we need to allocate several times as many threads as the number of CUDA cores.

Incidentally, the configuration of CUDA cores and memory per SM has been changing with each GPU generation, and the amount of each part per SM tends to increase. For example, the first generation GTX280 from 2005 had only 8 CUDA cores and 16 KBytes of shared memory per SM. This change in configuration changes the number of threads and blocks that can be efficiently allocated, so some trial and error is needed to determine the appropriate number of allocations for each GPU generation and computational target.

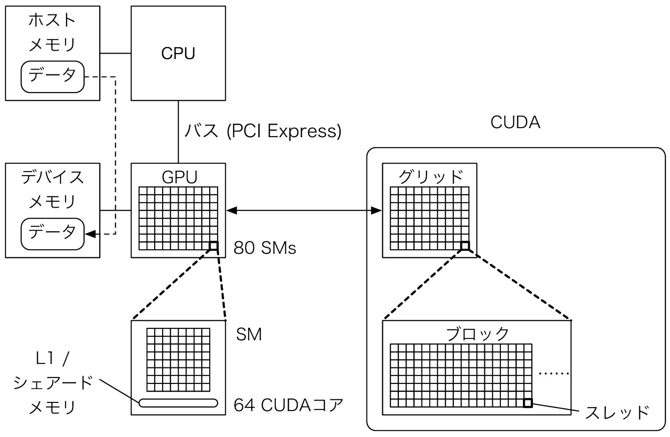


Fig. 15.1 Conceptual diagram of GPU and CUDA

CUDA uses such a programming model to execute the same instruction in multiple threads. This type of parallel processing is called **Single Instruction Multiple Thread** (**SIMT**) (Chapter 16). Multiple threads execute the same instruction.

In CUDA threads, it is assumed that the same calculation is basically performed on different data. For example, when using an if statement in a program for conditional branching, the then clause and the else clause can be executed in separate threads in parallel on the CPU. On the other hand, in CUDA, while one clause is being executed, the thread of the other clause is stopped. This processing method is one of the features of SIMT.

In addition, the GPU has a separate memory (**device memory**) for the GPU, which is different from the CPU's memory (called **host memory** to clarify the distinction). For this reason, it is necessary to explicitly copy data between the host memory and the device memory as needed (Figure 15.1). Since the data is copied via the PCI Express bus, this may become a bottleneck if the data is copied frequently. Therefore, ideally, once the data is passed to the GPU, it should not be returned to the CPU again and the calculation should continue. However, since the computation result data needs to be output to a file, it needs to be sent from the device memory to the host memory in the end.

15.2 Parallel computing of neurons

As before, let's write the code for parallelization on the GPU with the policy of assigning the computation of one neuron to one thread. The code is `random_gpu.cu`.

The differences between the CPU version (`random_ell.c`) are listed from the top. First, in lines 41 and 42, we define the constants `BLOCK_SIZE` and `GRID_SIZE`. These are the number of threads per block and the number of blocks required to compute N neurons, respectively, which will be explained in detail later.

Listing 15.1 `random_gpu.cu`

```
41 #define BLOCK_SIZE ( 32 )
42 #define GRID_SIZE  ( ( ( N ) + ( BLOCK_SIZE ) - 1 ) / ( BLOCK_SIZE ) )
```

The next step is to allocate the array in lines 53–56.

Listing 15.2 `random_gpu.cu:initialize`

```
53   cudaMallocManaged ( &n -> v,  N * sizeof ( float ) );
54   cudaMallocManaged ( &n -> ge, N * sizeof ( float ) );
55   cudaMallocManaged ( &n -> gi, N * sizeof ( float ) );
```

```
cudaMallocManaged ( &n -> s,  N * sizeof ( bool ) );
```

In the CPU version, this was a normal `calloc`. As explained above, the GPU memory and CPU memory are independent and need to be allocated separately. A function `cudaMalloc` is used, and must be called in parallel with `malloc` / `calloc`. In addition, the `cudaMemcpy` function must be called explicitly to copy data, but V100 *1 allows you to do so without being aware of it. When memory is allocated with the `cudaMallocManaged` function, it can be used as both host memory and device memory with the same variable name. Moreover, data is automatically copied as necessary to ensure consistency between the host and device. Such a system is called **unified memory**. However, it is very convenient because there is no need to manage the memory by oneself, so it should be used if the performance degradation is acceptable. Use `cudaMallocManaged` to allocate the array of the concatenation matrix as well.

Listing 15.3 `random_gpu.cu:initialize`

```
  cudaMallocManaged ( &n -> w,  n -> nc * N * sizeof ( float ) );
  cudaMallocManaged ( &n -> wc, n -> nc * N * sizeof ( int32_t ) );
```

The allocated unified memory is released by `cudaFree` at the end.

Listing 15.4 `random_gpu.cu:finalize`

```
  cudaFree ( n -> v );
  cudaFree ( n -> ge );
  cudaFree ( n -> gi );
  cudaFree ( n -> s );
  cudaFree ( n -> w );
  cudaFree ( n -> wc );
```

Let's take a look at the core: `calculateSynapticInputs` is a function that looks like the following:

Listing 15.5 `random_gpu.cu:calculateSynapticInputs`

```
__global__ void calculateSynapticInputs ( network_t *n )
{
  int32_t i = threadIdx.x + blockIdx.x * blockDim.x;

  if ( i < N ) {
    float re = 0, ri = 0;
    for ( int32_t j = 0, k = 0; ( k = n -> wc [ j + n -> nc * i ] ) != -1; j++ ) {
      float r = n -> w [ j + n -> nc * i ] * n -> s [ k ];
      if ( k < N_E ) { re += r; } else { ri += r; }
    }
    n -> ge [ i ] = exp ( - DT / TAU_E ) * n -> ge [ i ] + re;
    n -> gi [ i ] = exp ( - DT / TAU_I ) * n -> gi [ i ] + ri;
  }
}
```

The leading `__global__` indicates that the function is to be called by both the host and the device. Functions with `__global__` cannot return any value, so the function declaration must be `void`. And `for` loop on variable `i`:

Listing 15.6 `random_ell.c:calculateSynapticInputs`

```
  for ( int32_t i = 0; i < N; i++ ) {
```

is gone, and is rewritten as follows:

*1 GPU with Compute Capability 6 or higher to be precise.

Listing 15.7 `random_gpu.cu:calculateSynapticInputs`

```
  int32_t i = threadIdx.x + blockIdx.x * blockDim.x;

  if ( i < N ) {
```

The meaning of this will be explained later, along with the explanation of `BLOCK_SIZE` and `GRID_SIZE`, but now a unique number `i` is assigned to all threads issued. This is the same as the CPU version, but in general, more threads are issued than are needed (number of neurons), so only the necessary threads, i.e., those with thread numbers smaller than `N`, perform the actual calculation of synaptic input. Since the computation for each thread is done independently, 4000 computations are done simultaneously in this part.

The function `updateCellParameters` is similar. The function is prefixed with `__global__`, and the thread number is set at the beginning instead of the `for` loop at the beginning.

The loop part of the code looks like the following:

Listing 15.8 `random_gpu.cu:loop`

```
void loop ( network_t *n )
{
  timer_start ( );

  for ( int32_t nt = 0; nt < NT; nt++ ) {
    calculateSynapticInputs <<< GRID_SIZE, BLOCK_SIZE >>> ( n );
    updateCellParameters <<< GRID_SIZE, BLOCK_SIZE >>> ( n );
    cudaDeviceSynchronize ( );
    outputSpike ( nt, n );
  }

  double elapsedTime = timer_elapsed ( );
  printf ( "Elapsed␣time␣=␣%f␣sec.\n", elapsedTime);
}
```

The following line in the original code:

Listing 15.9 `random_ell.c:loop`

```
    calculateSynapticInputs ( n );
```

is replaced by:

Listing 15.10 `random_gpu.cu:loop`

```
    calculateSynapticInputs <<< GRID_SIZE, BLOCK_SIZE >>> ( n );
```

(The same is true for the `updateCellParameters`). This is a call to a function (called a **kernel**) on the GPU. When the kernel is invoked, it automatically creates the required number of threads, assigns a unique number to each thread, and each thread starts its own computation. The following description:

```
<<< GRID_SIZE, BLOCK_SIZE >>>
```

determines the number of threads, where `GRID_SIZE`×`BLOCK_SIZE` threads are issued. Now, the value of `BLOCK_SIZE` is 32, and that of `GRID_SIZE` is defined as

Listing 15.11 `random_gpu.cu`

```
#define GRID_SIZE  ( ( ( N ) + ( BLOCK_SIZE ) - 1 ) / ( BLOCK_SIZE ) )
```

which is the ceil of `N` divided by `BLOCK_SIZE`. In the case of $N = 4000$, `GRID_SIZE` = 125, so exactly `GRID_SIZE`×`BLOCK_SIZE` = 4000. If the number of neurons is not divisible by `BLOCK_SIZE`, then threads are issued in multiples of `BLOCK_SIZE` and as many threads as are closest to the number of neurons. the value of `BLOCK_SIZE` is arbitrary, but there are restrictions such as 1024 or less depending on the type of GPU. The value of `BLOCK_SIZE` is arbitrary, but there are restrictions such as 1024 or less depending on the type of GPU. The computation time varies depending on the value of `BLOCK_SIZE`, so we often try various values and adopt the fastest one. The former `g` in `<<< g, b >>>` is called the grid size, and the latter `b` is called the block size. Then, the next line follows:

Listing 15.12 `random_gpu.cu:loop`

```
    cudaDeviceSynchronize ( );
```

Since the kernel execution is asynchronous with the computation on the CPU side, if nothing is done, spikes will be output to a file on the CPU side while the GPU side is calculating the membrane potential. `cudaDeviceSynchronize` function explicitly waits for the end of kernel execution, so you can safely output the file.

Finally, let's also allocate the variable `n` in the unified memory.

Listing 15.13 `random_gpu.cu:main`

```
  cudaMallocManaged ( &n, sizeof ( network_t ) );
```

Be sure to `cudaFree` when the calculation is finished (line 174).

Let's try to do this.

```
DGX-Station:~/snsbook/code/part3/random$ make random_gpu
nvcc -O3 -I ../misc/SFMT-src-1.5.1/ -D SFMT_MEXP=19937 -c random_gpu.cu
gcc -O3 -std=gnu11 -Wall -I../misc/SFMT-src-1.5.1/ -D SFMT_MEXP=19937 -o SFMT.o -c ../misc/SFMT-
    src-1.5.1/SFMT.c
gcc -O3 -std=gnu11 -Wall -o timer.o -c ../misc/timer.c
nvcc -O3 -I../misc/SFMT-src-1.5.1/ -D SFMT_MEXP=19937 -o random_gpu random_gpu.o SFMT.o timer.o -
    lm
DGX-Station:~/snsbook/code/part3/random$ ./random_gpu
Elapsed time = 0.598377 sec.
```

The calculation was completed in about 0.60 s.

15.3 Parallel computing of synaptic inputs

Even though the V100 has 5120 cores, the number of neurons is 4000, so 1120 cores are not used and are idle. This is a waste. Generally, hundreds of thousands to millions of threads are issued at the same time when computing on a GPU. Let's consider further speedup by using such a huge number of threads.

If the number of synapses per post-side neuron is c, then the usual `for` loop calculation would involve $O(c)$ sum-of-products calculations, but this can be reduced to $O(\log c)$. This kind of calculation is called **reduction**. The conventional kernel was renamed `calculateSynapticInputs_a`, and version b was created by modifying it.

Listing 15.14 `random_gpu_reduction.cu:loop`

```
  for ( int32_t nt = 0; nt < NT; nt++ ) {
    calculateSynapticInputs_a <<< GRID_SIZE, BLOCK_SIZE >>> ( n );
    //calculateSynapticInputs_b <<< N, NTHSYN >>> ( n );
    updateCellParameters <<< GRID_SIZE, BLOCK_SIZE >>> ( n );
    cudaDeviceSynchronize ( );
    outputSpike ( nt, n );
  }
```

When the program is compiled and executed in this state, the computation completes in about 0.60 s, as in the previous chapter. Kernel b issues $N \times$ `NTHSYN` ≈ 8.2 million threads, where `NTHSYN` $= 2048$, and `#define` on line 43. Since kernel a issued only N threads, the number of threads is more than

2000 times larger than that. The ability to issue such a huge number of threads for computation is the real appeal of GPUs.

Let's take a look at the `calculateSynapticInputs_b`. This is the basis for everything. The contents are as follows.

Listing 15.15 `random_gpu_reduction.cu:calculateSynapticInputs_b`

```
__global__ void calculateSynapticInputs_b ( network_t *n )
{
  int32_t _i = threadIdx.x + blockIdx.x * blockDim.x;

  int32_t i = _i / NTHSYN;
  int32_t j = _i % NTHSYN;

  __shared__ float s_re [ NTHSYN ], s_ri [ NTHSYN ];

  if ( i < N ) {
    int32_t l = ( j < n -> nc ) ? n -> wc [ j + n -> nc * i] : -1;
    s_re [ j ] = ( l != -1 && l < N_E  ) ? n -> w [ j + n -> nc * i ] * n -> s [ l ] : 0.;
    s_ri [ j ] = ( l != -1 && l >= N_E ) ? n -> w [ j + n -> nc * i ] * n -> s [ l ] : 0.;
    for ( int32_t k = NTHSYN; k < n -> nc; k += NTHSYN ) {
      int32_t l = ( j + k < n -> nc ) ? n -> wc [ j + k + n -> nc * i] : -1;
      s_re [ j ] += ( l != -1 && l < N_E )  ? n -> w [ j + k + n -> nc * i ] * n -> s [ l ] : 0.;
      s_ri [ j ] += ( l != -1 && l >= N_E ) ? n -> w [ j + k + n -> nc * i ] * n -> s [ l ] : 0.;
    }
  }
  __syncthreads();

  for ( int32_t k = NTHSYN / 2; k > 0; k >>= 1 ) {
    if ( j < k ) {
      s_re [ j ] += s_re [ j + k ];
      s_ri [ j ] += s_ri [ j + k ];
    }
    __syncthreads();
  }

  if ( i < N && j == 0) {
    n -> ge [ i ] = exp ( - DT / TAU_E ) * n -> ge [ i ] + s_re [ 0 ];
    n -> gi [ i ] = exp ( - DT / TAU_I ) * n -> gi [ i ] + s_ri [ 0 ];
  }
}
```

The idea is this: on the GPU, threads are not treated independently, but together in a unit called a **thread block**; the number of threads in a thread block is the block size, which is specified by the value of b in `<<< g, b >>>`. In our code, `NTHSYN` $= 2048$, which means that 2048 threads act as one block. The number of threads issued by this code was $N \times 2048$. This means that for each post-side neuron, 2048 threads are used to calculate the synaptic input [*2].

First, we need to determine the number i of the post-side neuron that each thread is responsible for and the local thread number j in

Listing 15.16 `random_gpu_reduction.cu:calculateSynapticInputs_b`

```
  int32_t _i = threadIdx.x + blockIdx.x * blockDim.x;

  int32_t i = _i / NTHSYN;
  int32_t j = _i % NTHSYN;
```

The next step is to allocate shared memory.

[*2]The number 2048 was determined based on the criterion of several times the number of CUDA cores.

Listing 15.17 `random_gpu_reduction.cu:calculateSynapticInputs_b`

```
140  __shared__ float s_re [ NTHSYN ], s_ri [ NTHSYN ];
```

Shared memory is memory that can be shared by threads in the same block. Although shared memory has a small capacity, it can be accessed much faster than device memory [*3] and is effective for speeding up the process if it can be used properly. The shared memory is declared with the prefix `__shared__` and stores the progress of the calculation of `re` and `ri`.

Since we have allocated shared memory, we will use it to proceed with the sum-of-products calculation.

Listing 15.18 `random_gpu_reduction.cu:calculateSynapticInputs_b`

```
142  if ( i < N ) {
143    int32_t l = ( j < n -> nc ) ? n -> wc [ j + n -> nc * i] : -1;
144    s_re [ j ] = ( l != -1 && l < N_E  ) ? n -> w [ j + n -> nc * i ] * n -> s [ l ] : 0.;
145    s_ri [ j ] = ( l != -1 && l >= N_E ) ? n -> w [ j + n -> nc * i ] * n -> s [ l ] : 0.;
146    for ( int32_t k = NTHSYN; k < n -> nc; k += NTHSYN ) {
147      int32_t l = ( j + k < n -> nc ) ? n -> wc [ j + k + n -> nc * i] : -1;
148      s_re [ j ] += ( l != -1 && l < N_E )  ? n -> w [ j + k + n -> nc * i ] * n -> s [ l ] : 0.;
149      s_ri [ j ] += ( l != -1 && l >= N_E ) ? n -> w [ j + k + n -> nc * i ] * n -> s [ l ] : 0.;
150    }
151  }
152  __syncthreads();
```

Using all 2048 threads for each neuron, thread `j` calculates the product of w_{ij} and s_j in lines 144 and 145, and places it at position `j` in shared memory. Next, if the number of synaptic connections is greater than 2048, each thread adds the value of the product of w_{ij} and s_j to the shared memory (lines 146–150), but since the current number of synapses is less than 2048, this loop does not turn. Finally, we call the `__syncthreads` function to synchronize the threads and wait for all threads to finish their calculations.

Listing 15.19 `random_gpu_reduction.cu:calculateSynapticInputs_b`

```
140  __syncthreads ( );
```

Up to this point, we have been able to place all the values necessary for the calculation in the shared memory. When using shared memory, the first step is to place the necessary values.

The next step is to calculate the sum of the 2048 values on the shared memory.

Listing 15.20 `random_gpu_reduction.cu:calculateSynapticInputs_b`

```
154  for ( int32_t k = NTHSYN / 2; k > 0; k >>= 1 ) {
155    if ( i < N && j < k ) {
156      s_re [ j ] += s_re [ j + k ];
157      s_ri [ j ] += s_ri [ j + k ];
158    }
159    __syncthreads ( );
160  }
```

This calculation can be easily seen in Fig. 15.2, which shows how the function `__syncthreads` by using half of the 2048 threads, 1024, to add its own value and the values of the 1024 threads ahead. This is the first loop. In this way, the sum of 2048 values can be reduced to the sum of 1024 values. In the second loop, we use half the number of threads, 512, to add our value and the 512 values ahead of us, and call `__syncthreads` again. Then, the sum of 1024 values becomes the sum of 512 values. In this way, the number of elements used to calculate the sum is halved, and the process is

[*3]The access speed performance is shared memory (SRAM) $\gg$ device memory (GDDR 5) $\gg$ host memory (DDR 4).

repeated until the final result is a single value [*4]. By computing in parallel in this way, the number of loops is reduced to only $O(\log_2 2048) = 11$.

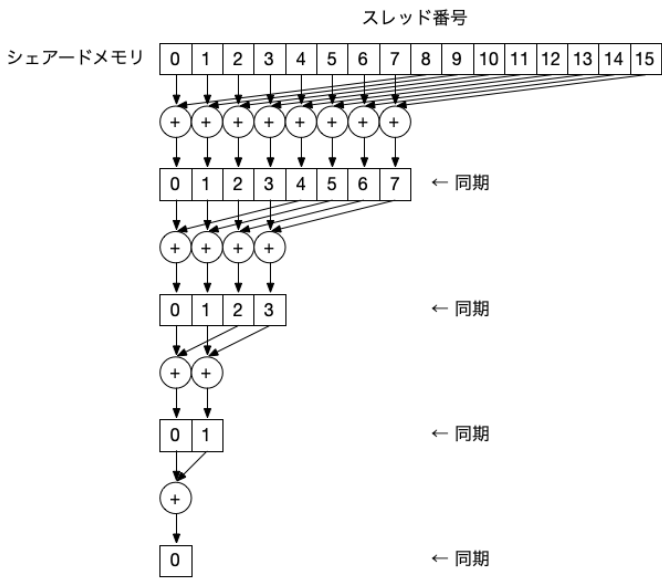


Fig. 15.2 Computation of reduction using shared memory. An example with 16 threads is shown.

Finally, the representative of the 2048 threads writes the calculation results to the device memory. In this example, the representative is thread 0.

Listing 15.21 `random_gpu_reduction.cu:calculateSynapticInputs_b`

```
if ( i < N && j == 0 ) {
  n -> ge [ i ] = exp ( - DT / TAU_E ) * n -> ge [ i ] + s_re [ 0 ];
  n -> gi [ i ] = exp ( - DT / TAU_I ) * n -> gi [ i ] + s_ri [ 0 ];
}
```

That's it for the code. Let's see how fast this program is by commenting out the kernel for a and enabling the kernel for b instead.

```
DGX-Station:~/snsbook/code/part3/random$ make random_gpu_reduction
nvcc -O3 -I../misc/SFMT-src-1.5.1/ -D SFMT_MEXP=19937 -c random_gpu_reduction.cu
nvcc -O3 -I../misc/SFMT-src-1.5.1/ -D SFMT_MEXP=19937 -o random_gpu_reduction
     random_gpu_reduction.o SFMT.o timer.o -lm
DGX-Station:~/snsbook/code/part3/random$ ./random_gpu_reduction
Elapsed time = 0.220382 sec.
```

In this case, it was more than twice as fast. If the number of synapses per neuron is larger, the difference from the kernel of a becomes even larger.

As you can see, GPU needs to be written in a way that considers its architecture, but hopefully the performance will be commensurate with it, and there may be ways to make it even better. many documents on GPUs are available on the net, and you can refer to them [*5]. In fact, this reduction is an almost direct application of NVIDIA's Optimizing Parallel Reduct ion in CUDA (Harris, 2007) to the problem at hand, and this document contains further speedup techniques.

[*4]Considering warp divergence and memory bank conflicts, pull it to the left side.

[*5]For example, the NVIDIA Programming Guide is a very good resource (`https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`).

Chapter 16

Other parallel computing methods and speed-up techniques

In this book, we have introduced the basics of how to perform parallel computations on neurons and synapses using OpenMP, MPI, and GPUs. There are various other parallel computation methods and speed-up techniques, but in this document, we have limited ourselves to the basics and omitted the more complex cases. In this section, we briefly introduce more advanced parallel computations that we have not been able to introduce so far, and introduce related references, which we hope you will find useful in your own studies. In particular, 片桐 (2013) is a good book that covers the basics and explains them very clearly.

16.1 SIMD

There is a parallelization technique called **SIMD** (**Single Instruction Multiple Data**). In SIMD, there are special registers of 512 bits that can store 8 double (64 bit) variables or 16 float (32 bit) variables and perform the same operation to all the variables. In the LIF model, the membrane potential decay of 16 neurons can be done in a single operation. This kind of calculation is called **vector computation**. SIMD was first developed in the 1960s with the ILLIACIV (Slotnick, 1982), a SIMD unit with a vector length of 64 bits, and the vector length has gradually increased to 512 bits in CPUs of the 2020s. The A64FX, the CPU of Fugaku, has a SIMD mechanism called **SVE** (**Scalable Vector Extension**), which provides special registers with a length of 512 bits. Intel's recent CPUs also have a similar mechanism called AVX (Advanced Vector Extension), which provides 256-bit or 512-bit long registers depending on the type of CPU.

Since the example program in this book does not use SIMD, the vector calculation that is originally possible is not performed, and the performance is reduced to one eighth of the vector length. Ideally, SIMD should be used, but in reality, its usage differs considerably depending on the computing environment, and we do not dare to use it this time because we think it is not suitable for beginners.

There are many ways to compute neurons in parallel using SIMD, but the main method is to assign state variables of multiple neurons and synapses to each element of a vector and compute them in parallel. Normally, a SIMD unit is installed in the CPU, and SIMD parallelism can be used simultaneously with CPU parallelism using MPI and CPU core parallelism using OpenMP, resulting in three levels of parallelism when combined with hybrid parallelism. For example, if an eight-core CPU is used for OpenMP thread parallelism, two CPUs are used for MPI process parallelism, and two SIMD arithmetic units of 512-bit vector length are used for each CPU core for SIMD parallelism in single precision, then $8 \times 2 \times 2 \times 16$ elements can be calculated simultaneously.

There are four main ways to implement SIMD: using a machine language called assembly, using special functions called built-in functions, using automatic parallelization by a compiler, and using the OpenMP SIMD description method. There are four main ways to perform SIMD: using machine language called assembly, using special functions called built-in functions, using automatic parallelization by a compiler, and using OpenMP's SIMD description method. All of these methods have their advantages and disadvantages, and which one is best depends on the computing environment and the target program. In addition to the OpenMP SIMD description method, there are other methods that depend on the computing environment. The following is a brief description of the features of each.

Auto-parallelization is the simplest way to compile and run a program by selecting the compile option that best suits the computing environment. The disadvantage is that the compiler often fails to recognize the parts of the program where SIMD can be used, resulting in low SIMD utilization. The degree of this recognition depends on the complexity of the program and the compiler. Anyway, this is an easy method to try, so it's worth a try at first.

Assemblies and built-in functions are a way to specify all the parts of a SIMD calculation by yourself. Although this method requires more time and effort for programming, it is a reliable way to perform SIMD calculations. However, since the description method differs depending on the CPU and programming language, it is necessary to respond to each individual case.

SIMD programming with OpenMP was introduced in OpenMP 4.0 released in the 2013s, and is a method that can be executed with the same program code in the corresponding computing environment. It has the advantage of being easy to use with just a few lines of instructions and of being able to standardize the program code across computing environments, and thus has the potential to be established as a platform for SIMD in the future.

Finally, I would like to explain the classification of parallel computing in relation to the name SIMD. The name SIMD was proposed for a classification of parallel processing architectures called **Flynn's classification** (Hennessy and Patterson, 2017). There are also **MIMD** (**Multiple Instruction Multiple Data**), SISD, MISD, and others. MIMD is an architecture that executes multiple instructions on multiple data, and is equivalent to a multi-core CPU. On the other hand, there is a term **SIMT** (**Single Instruct ion Multiple Threads**) that refers to the parallel computing method on GPUs (see Chapter 15), since multiple threads execute a single instruction simultaneously in GPU-based computing. **SPMD** (**Single Program Multiple Data**) refers to the execution of multiple data in a single program. SPMD is the format used in OpenMP (Chapter 13) and MPI (Chapter 14), where each thread in OpenMP or each process in MPI runs the same program and processes different data. Of course, there can also be parallelization called MPMD (Multiple Program Multiple Data). For example, a program that models the cerebral cortex and a program that models the cerebellum are executed simultaneously while communicating with each other.

16.2 Pipeline processing

Pipeline processing is a form of processing in which the processing involved in an operation is divided and the operation is performed in the form of a bucket relay or factory line processing (Hennessy and Patterson, 2017). Simply put, to execute pipeline processing efficiently, the number of rotations of the innermost loop, which is the bottleneck of the computation, should be secured sufficiently, and the computation in the loop should be simplified as much as possible. These two things are necessary to fully fill the pipeline with processing.

For example, in a neural circuit simulation, if the calculation of synaptic conductance involves a loop with a synapse number at the innermost part of the loop, it is necessary to make the number of synapses rotating in the loop sufficiently large and to simplify the processing by eliminating if statements in the loop as much as possible. In this case, the number of synapses to be rotated in the loop is sufficiently large. A more advanced method is called **loop unrolling**, in which the loop is manually expanded to increase the utilization rate of the pipeline.

16.3 Cache memory

In 2020s computing systems, memory speed is relatively slow compared to computation speed, and the gap is unlikely to be eased in the future, but is likely to worsen. Therefore, caches will continue to be needed to improve efficiency. The efficient use of cache requires an understanding of the basic nature of cache (Hennessy and Patterson, 2017). I will briefly explain its essence. The idea is simple: put frequently used data on the cache memory and use it as many times as possible. For example, a state variable of synaptic conductance can be placed in cache memory and used over and over again in an arithmetic unit. If it were possible, memory access would be many times faster. However, it is easier said than done. In practice, it is not so simple. The size of cache memory (KBytes – MBytes) is less than 1/1000 of the size of main memory (GBytes), and the exchange of data between main memory and cache memory is automated by hardware. In normal processing, the data coming from the main memory quickly exceeds the size of the cache, so it is returned to the main memory one after another when the processing is finished. For example, if there are 1 GBytes of synaptic conductance data in the main memory, even if it is necessary to bring all the data to the cache memory and calculate them in order, it is not possible to bring all the data at once because it does not fit in the cache memory capacity. Therefore, it is necessary to send the data from the main memory to the cache little by little, and when the calculation is completed, return it to the main memory each time, and then send the next portion little by little.

Efficient use of the cache means that once the data is in the cache, it is used for many operations before being returned to the main memory, thereby reducing the slow main memory access. This is controlled indirectly by the way the program is processed, and it is not that simple.

A typical method to improve cache efficiency is **cache blocking**, which is a method to adjust the data structure and looping method according to the size of the cache. It depends on the specifications of the computer system and the parameters of the neural circuit model.

16.4 Communication

MPI communication takes several microseconds level of communication time, which is 10–1000 times longer than other processes such as arithmetic (a few nanoseconds) or memory access processing (a few hundred nanoseconds). Nevertheless, the example program presented in this book is a case of 24 MPI processes, and the scale of the communicating processes is small, so the time required for communication is negligible compared to the computation time of 4000 synaptic connections. However, when tens of thousands of MPI processes are used, such as in a supercomputer, the overhead of communication and synchronization between tens of thousands of processes becomes large, and the communication time becomes large, so there is a need to improve the efficiency of communication.

In this section, we will introduce a method for reducing the number of communications in neural circuit simulations. In the case of synapses, there is a signal propagation delay of about 1 ms or more between the onset of firing in the cell body and the generation of synaptic currents by synaptic transmitters. In contrast, when simulating with the HH model, the time step is only 0.01 ms, or 1/100 of the delay time. Therefore, the synaptic current is generated 100 steps after the step where the fire occurs. If we communicate at the step where the firing occurs, there is no need to communicate during the 99 delayed steps. In this way, the frequency of MPI communication during the delay time can be greatly reduced. If this is extended to the entire neural network, it is possible to perform MPI communication of firing information together at each minimum signal propagation delay of the network (about 1 ms). This method has been adopted by leading neural circuit simulators such as NEURON and NEST, and is particularly effective in large-scale neural circuit simulations. In addition, there is a method to reduce the communication latency by executing

MPI communication and computation simultaneously. In order to do so, it is necessary to adjust the timing of communication and computation using MPI asynchronous communication, but this is too complicated to deal with in this introduction, so it is omitted in this article.

These MPI-related speedups are also the subject of research in computational neuroscience in the 2020s, and they are changing day by day. MPI is likely to be used in many cases of large-scale neural circuit simulation on supercomputers, but such cases and the population itself are not large to begin with, so various information tends to be insufficient. However, since the number of such cases and the population itself is not large, there tends to be a lack of various information. Even if you cannot find information on the Internet or in books, modern computers have a variety of documents and specifications, so the only way to find out is to diligently study them.

16.5 Column: AoS vs. SoA

If you are a clever code writer, you must have seen the code for the HH model (Listing 3.1) and thought, "Why not make it a structure?" In other words, instead of declaring the variable naked as:

```
double v, m, h, n;
```

declere the following `struct`:

```
typedef sturuct {
  double v, m, h, n;
} hh_t;
```

and define the structure of the neuron, the variable declaration will be:

```
hh_t neuron;
```

Moreover, when you increase the number of neurons, you can simply write:

```
hh_t neuron [ N ];
```

It can be written as access to variables, such as the membrane potential v of neuron i, can be handled by `neuron[i].v`. This implementation method has a name. Since it defines an array of structures, it is called **Array of Structure** (**AoS**). On the other hand, the method of preparing an array of neurons in a single structure is called **Structure of Array** (**SoA**). Specifically, we define the following `struct`:

```
typedef struct {
  double v [ N ], m [ N ], h [ N ], n [ N ];
} hh_t;
```

and declare variables as:

```
hh_t neuron;
```

In this case, we can handle the membrane potential variable with `neuron.v[i]`.

Now, both AoS and SoA do the same calculations. In terms of code abstraction, AoS seems to make more sense. On the other hand, SoA is known to be faster when accelerators such as GPUs are used.

Let's try it out. `make` under `code/column/soa/` will generate `aos` and `soa`. The content is a simulation of 65536 independent HH models for 1s on GPU.

When run on my GPU (NVIDIA Tesla V100), AoS completed the calculation in about 1 second, while SoA completed it in about 0.67 seconds. This is a speedup of about 1.5 times.

Why does this happen? In AoS, for each of the N neurons, v, m, h, and n are placed in order (Figure 16.1A). When computing on the GPU, the variable v is accessed first, and the computation is performed on the value of v at each jumping address. The GPU first accesses the variable v, and the computation is done on the values of v that exist at the jumping addresses. This makes it difficult to store all the necessary information in the cache memory because memory access is inefficient since it covers the entire memory space. Such a pattern of accesses to jumping addresses is called **stride access**. In SoA, on the other hand, the values of N neurons are placed in order for each variable (Figure 16.1B). In this case, each thread accesses consecutive addresses. Since memory access is localized, it can be accessed efficiently and stored well in the cache memory.

In summary, it is necessary to choose the data structure considering the hardware used for the simulation. The NEURON simulator was originally written in AoS. In 2019, the SoA version was finally completed and released under the name CoreNEURON (Kumbhar et al., 2019). On the other

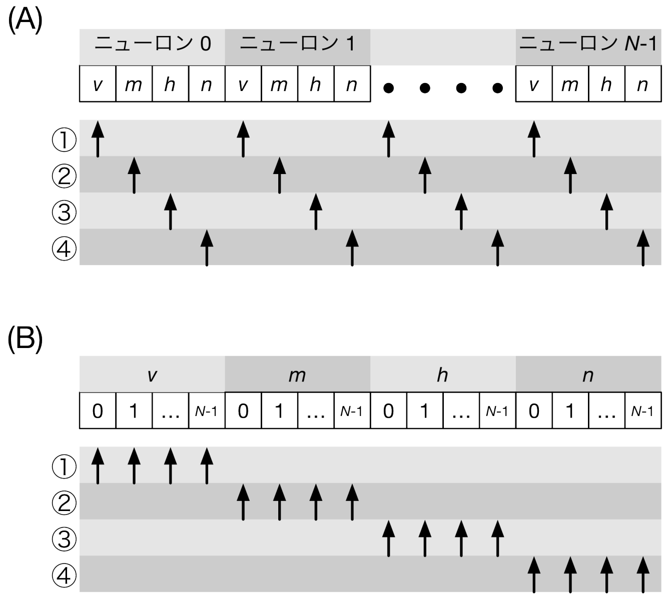


Fig. 16.1 Differences in memory access patterns between AoS and SoA. A: In AoS, the member variables of the structure v, m, h, n are placed in memory in order from neuron 0 to $N-1$. The accesses of the threads of GPU are indicated by arrows, where ① to ④ indicate the order of access. B: The same applies to SoA.

hand, another simulator, Arbor (Akar et al., 2019), which aims to be compatible with the NEURON simulator, was written in SoA from the beginning, anticipating the use of GPUs.

Thus, fast execution may require changes in the program, such as data structures, which themselves take time. When making changes to one's own program, the decision should be made after weighing the development time and execution time of the program. For example, a neural circuit simulation program that takes one minute to run once can be run in two hours if it is run 100 times. Taking a week to speed up this program by a factor of 10 is not very meaningful (although it is good to do it as a study). It would be faster to wait for two hours with the slow program. On the other hand, if we were to speed up this program and use it to control a robot, it would be necessary to complete a one-second simulation within one second of computation time, so-called real time execution. In this case, it is essential to achieve real time execution, so it is worthwhile to spend a week or even a month.

Part IV

Forefront of Spiking Network Simulation

Throughout Parts I–III, we have provided a broad introduction to the fundamentals and applications of spiking network simulation, including the basics of neuroscience and computational science. Readers who have worked with their hands up to this point are already in a position to actually write code and start research in this field. On the other hand, in order to actually get started, it is necessary to understand the background and prospects of the field: how it has evolved over the years, what projects are currently underway, and how it will develop in the future. This is summarized in Part IV.

Chapter 17

Past, present, and future of spiking network simulation

How has spiking network simulation progressed so far, and where is it headed in the future?

17.1 Current status of spiking network simulation in the world

17.1.1 Blue Brain Project[*1] (2005–)

To the best of our knowledge, one of the major trends in neural circuit simulation, especially large-scale simulation, originated in the Blue Brain Project (Markram, 2005) initiated by Henry Markram at the Ecole Polytechnique Fédérale de Lausanne (EPFL) in Switzerland. Using the Blue Gene supercomputer developed by IBM [*2], the Blue Brain Project aims to build a digital "brain" that incorporates all the detailed knowledge of the brain's neural circuits, and to use it to understand brain function and its disorders. The abstract of the paper says "from first principles," meaning that the construction is strictly based on experimental facts and not on the subjectivity of the researcher (Markram, 2005).

Perhaps the best example of the results of the Blue Brain Project is the paper "Reconstruct ion and Simulat ion of Neocortical Microcircuit ry" published in Cell in 2015 [*3] (Markram et al., 2015). This paper, with 82 (!) co-authors, reconstructed a functional unit corresponding to about 0.3 mm^3 of the rat primary somatosensory cortex, taking into account 55 different morphologies and 207 different electrical properties. It was precisely reproduced with 31,000 multi-compartment neurons and 37,000,000 synapses. The morphology and electrical parameters of the cells are all based on actual experimental data, resulting in a highly accurate reproduction. In addition, we also performed numerical simulations to observe the behavior of the entire network, such as the rapid transition from synchronous to asynchronous activity. The NEURON simulator (Carnevale and Hines, 2006)was used for the numerical simulation, and the IBM Blue Gene series supercomputer was used.

Following the active activities of the Blue Brain Project, the Human Brain Project was launched in 2013 as a huge EU-wide project.

[*1] `https://www.epfl.ch/research/domains/bluebrain/`

[*2] It is customary for IBM to add the prefix "Blue" to its product names, and the Blue Brain Project seems to have followed this practice.

[*3] Open access, so anyone can read it.

17.1.2 Large-scale simulation of cortical-thalamic model (2007)

Eugine Izhikevich, then at the University of California, San Diego, developed the Izhikevich model (Section 3.3.1) to simulate a large-scale cortical-thalamic model (Izhikevich and Edelman, 2008). 1 million spiking neurons of 22 types with several compartments were arranged based on the laminar organization of the cerebral cortex, and STDP was implemented. Spontaneous rhythmic activity was reproduced. In the appendix of the paper, there is only one line stating that "human-scale simulations were also conducted. In the appendix of the paper, there is one line stating that "human-scale simulations were also conducted," but the details are not clear.

17.1.3 Simulation of a cat-scale corticothalamic model (2009)

Dharmendra S. Modha's group at IBM used their supercomputer, BlueGene/P, to conduct a spiking network model simulation of a cortical-thalamic model consisting of 1.6 billion neurons and 8.9 trillion synapses, and succeeded in reproducing neural activity equivalent to brain waves (Ananthanarayanan et al., 2009). The scale of the model is equivalent to the cerebrum of a cat. The paper was presented at SC'09, the world's largest international conference on supercomputers, and won the Gordon Bell Award that year. The title of the paper was catchy, and it attracted a lot of attention. This series of research led to the development of TrueNorth, a neuromorphic chip (see Section 17.3.3).

17.1.4 Functional model of the cerebral cortex using spiking neurons (2012)

Chris Eliasmith's group at the University of Waterloo (Canada) has built a functional model of the cerebral cortex consisting of 2.5 million spiking neurons, focusing on their function rather than on faithful reproduction of actual neural circuits (Eliasmith et al., 2012). This model is capable of performing complex tasks such as image recognition, working memory tasks, and even motor control of a robot arm. The framework used here, Nengo, will also be used as a front-end for the neuromorphic chip Loihi that will appear later.

17.1.5 Human Brain Project (2013–)

The European Union's Human Brain Project [*4] is a major neuroscience research project that aims at a comprehensive understanding of the brain and its medical and engineering applications, not only targeting the digital reconstruction of the brain as in the Brain Brain Project, but also more comprehensively targeting the human brain from rodents (Amunts et al., 2016). Since its inception, the Human Brain Project has been criticized by many inside and outside the project for its objectives and organizational structure, and for a while its existence was threatened, but it has gradually undergone organizational reforms and is now a more open project. However, it has gradually undergone organizational reforms and is now a more open project.

The Human Brain Project aims to digitize and publish all the findings of the project, and EBRAINS [*5] is being operated as an archive for this purpose. EBRAINS contains not only papers and data, but also many models and simulators that can be freely reused. The Human Brain Project is a 10-year project and will end in 2023, but EBRAINS will continue to operate independently.

For neural circuit simulation, the NEST simulator (Gewaltig and Diesmann, 2007) and the NEURON simulator (Carnevale and Hines, 2006) are mainly used, and their development is also underway. The NEST is used as a front-end for the SpiNNaker neuromorphic chip (van Albada et al., 2018), and the NEURON simulator is increasingly compatible with various accelerators including

[*4] `https://www.humanbrainproject.eu/en/`
[*5] `https://ebrains.eu`

GPUs (Kumbhar et al., 2019), playing a role in connecting experiments to applications.

17.1.6 BRAIN Initiative (2014–)

Following the launch of the Human Brain Project, the National Institutes of Health (NIH) and other organizations in the U.S. have been working on the development of **Brain Research through Advancing Innovative Neurotechnologies (BRAIN) Initiative** [*6] (Ramos et al., 2019). One of the core organizations of the BRAIN Inititative is the Allen Institute for Brain Science, which is a world center for the generation of neural wiring data (connectomes), and in particular for the generation of computational resources. The Allen Institute for Brain Science is a global center for the generation of neural wiring data (connectome), for which a vast amount of computational resources are used. In neural circuit simulation, a very accurate model of layer 4 of the primary visual cortex has been constructed, as has the Blue Brain Project [*7] (Arkhipov et al., 2018). Layer 4 of the primary visual cortex is the first layer where visual stimuli from the eye enter the brain via the lateral kneecap, and this model has been shown to respond very realistically to simulated visual stimuli.

The BRAIN Initiative is also actively promoting the release of data, and the Allen Brain Atlas [*8] is particularly comprehensive and overwhelming for rodents. In addition to the BRAIN Initiative, the collection and publication of connectome data is also being conducted on a large scale in the U.S. Human data in particular is collected and published by the Human Connectome Projects [*9]. These experimental data will become more important in the future for simulations of data assimilation, in which neurons are placed and connected faithfully.

17.1.7 Brain Mapping by Integrated Neurotechnologies for Disease Studies (Brain/-MINDS) (2014–)

In Japan, the Brain/MINDS [*10] project was launched in 2014 (Okano et al., 2016). The unique aspect of the Brain/MINDS is that it targets marmosets as model animals. The marmoset is a primate that exhibits higher brain functions and is capable of genetic modification. By taking advantage of this feature, we aim to elucidate the mechanism of higher brain functions, which in turn will contribute to overcoming human mental and neurological disorders and to the advancement of information processing technology. We are generating marmoset connectome data, and resources are being invested intensively in the analysis of the connectome of the prefrontal cortex, which is thought to be the source of higher brain functions. On the other hand, neural circuit simulation is not included in the project itself, and collaboration with external organizations is being promoted.

The Brain/MINDS also operates a portal site [*11], where a variety of data, from marmosets to gene atlases, tracer data, and MRIs, are available. In the past, it was very difficult to release experimental data due to rights issues and technical problems, but now, as in this project, the openness of scientific projects is rapidly progressing.

17.2 Current status of spiking network simulation in Japan

Returning to the topic of spiking network simulation, research using supercomputers has been actively conducted in Japan.

[*6] `https://braininitiative.nih.gov/`
[*7] This paper is also open access, so anyone can read it.
[*8] `https://portal.brain-map.org/`
[*9] `https://www.humanconnectome.org/`
[*10] `https://brainminds.jp/`
[*11] `https://dataportal.brainminds.jp/`

17.2.1 ISLiM (2006–2013)

At the same time as the Blue Brain Project was launched, a project for the simulation of life systems, "Research and Development of Integrated Simulation Software for Next Generation Life Forms (ISLiM)" [*12] was launched in Japan, led by RIKEN, based on the premise of using a petascale supercomputer (later called "K computer"). During the project period, the K computer was used to simulate brain neural circuits as well as cells and organs. During the project period, the K computer was officially put into operation (2012), giving a boost to the project.

One of the main features of this project is the "Insect Whole Brain Simulation" by Ryohei Kanzaki's lab at the University of Tokyo (宮本 et al., 2015). In this research, the morphological data of silkworm moth neurons and the simulation of single neurons were developed by Kanzaki's laboratory, and all silkworm moth neurons were described by a multi-compartment model, and a network was constructed and simulated using a supercomputer. In addition to the above, they also conducted a simulation of the olfactory-motor system of the silkworm moth. This was followed by a simulation of the olfactory-motor system of the silkworm moth, which led to Post-K Exploratory Challenge #4 from 2017. In addition, an ambitious textbook entitled "Building Insect Brains" has been written and published, and the research is being vigorously pursued with a vertically integrated and highly coherent story, from morphological data acquisition to silkworm moth robot control (神崎, 2018).

Another highlight was the benchmarking of a small monkey-scale brain neural circuit simulation using all K computer nodes, which was released in 2013 (Kunkel et al., 2014). The simulation of a neural circuit with 1.7 billion neurons randomly connected to 10 trillion synapses, which is equivalent to 1% of the human cerebral cortex in scale, was successfully performed using all nodes of the K computer. In addition, it took 40 minutes to simulate just one second of the brain. It took 40 minutes to simulate only one second of the brain. This result is a benchmark, and it does not improve our understanding of the information processing mechanism of the brain, but it plays a very important role in developing a roadmap for expanding the scale of the network and realizing human whole-brain simulation in the future.

In this project, a real time simulation of the spiking network model of the basal ganglia was also conducted using a GPU (Igarashi et al., 2011), in collaboration with the Honda Research Institute Japan, which has been conducting brain research since the ASIMO and other robotics research. This is a joint research with Honda Research Institute Japan, which had been conducting brain research from robotics research such as ASIMO, aiming to control the robot's behavioral selection with a basal ganglia model.

17.2.2 HPCI Strategic Program Area 1: Predictive Life Science, Medicine and Drug Discovery (2011–2016)

In parallel with ISLiM, the HPCI Strategic Program [*13] was launched. In the context of spiking network simulation, as a hierarchical integrated simulation for predictive medicine, i.e., a hierarchical integrated simulation of the cardiovascular system and the musculoskeletal and nervous systems, we conducted an integrated simulation of the nervous system-musculoskeletal system to reproduce the symptoms of Parkinson's disease. Specifically, we developed a hierarchically integrated simulator of the whole human body's neuro-musculoskeletal system from the nerve cell level and muscle fiber level to predict the pathophysiology of motor dysfunction in Parkinson's disease and to support treatment.

Parkinson's disease is a neurological disorder that increases in incidence around the age of 50 and affects millions of people worldwide. Parkinson's disease is often associated with motor disorders

[*12]Next-Generation Integrated Simulation of Living Matter, `https://www.csrp.riken.jp/`
[*13]`http://www.scls.riken.jp/`

such as tremors, motor delays, difficulty in movement, and difficulty in maintaining posture. The cause of the disease is a decrease in dopamine in the brain due to the loss of dopamine-producing neuronal cells in the substantia nigra, which is caused by genetic factors, disorders, and intracellular oxidative stress. The mechanism of the motor symptoms caused by this decrease in dopamine is not well understood. In this project, we focused on the mechanism of tremor generation and conducted a large-scale neural circuit simulation. The results suggested a mechanism for the generation of abnormal oscillatory neural activity in the beta frequency band (15Hz) in the basal ganglia of patients (Shouno et al., 2017), and oscillatory neural activity in the thalamus and cerebral cortex that causes tremors, as well as oscillatory neural activity in the alpha frequency band (五十嵐 et al., 2015). By coupling this brain model with a musculoskeletal model and running it on a Kyoto computer, we are also conducting simulations of tremors caused by brain activity in Parkinson's disease states. This shows the possibility of approaching neurological diseases by multiscale simulation of the body and brain.

17.2.3 Real-time simulation of cerebellar model (2011)

When Yamazaki, one of the authors of this book, heard about the real-time simulation of the basal ganglia, he thought that real-time simulation should be done in the cerebellum. This is because the cerebellum is responsible for motor control and motor learning, which require real-time sensing and action. From this point of view, Yamazaki and Igarashi, another author of this book, started collaborative research and developed a GPU version of the cerebellar model consisting of 100,000 neurons. To demonstrate the real-time nature of the system, we demonstrated a batting robot that simulates blink reflex conditioning (Section 6.3). We first presented our work at the Japanese Society for Neural Networks in 2011, and have been working with the robot ever since. In 2013, we co-authored a paper (Yamazaki and Igarashi, 2013). Since then, we have steadily increased the scale of the network, from 1 million neurons in real time using four GPUs (Gosui and Yamazaki, 2016), to 1 billion neurons in real time using the RIKEN supercomputer "Shoubu" (Yamazaki et al., 2019), and finally to Human-scale 68 billion neurons on the K computer (Yamaura et al., 2020).

17.2.4 Post-K exploratory challenge #4 (2017–2020)

A few years have passed since the end of ISLiM, but a neural circuit simulation project using the successor to the K computer, the current Fugaku, which was called the post-K computer at the time, started in 2017: "Elucidation of Neural Circuit Mechanisms for Thinking and Their Application to Artificial Intelligence. In this project, the research and development of a rodent whole-brain simulation + brain-type artificial intelligence led by Professor Kenji Doya of Okinawa Institute of Science and Technology (OIST) and an insect brain simulation led by Professor Ryohei Kanzaki of the University of Tokyo were adopted, and each conducted their own research.

The authors participated in the Doya group's rodent whole brain simulation [*14] and contributed to the promotion of the project. A number of results have already been obtained, but those that have been published include the following:

- Development of a high-performance neural circuit simulator MONET (Igarashi et al., 2019)
- Human-scale cerebellar simulation using the entire K computer (Yamaura et al., 2020)

In the meantime, the K computer was shut down in the summer of 2019, and its place was taken by its successor, the supercomputer Fugaku.

[*14] `https://brain-hpc.jp/postk`

Development of a high-performance neural circuit simulator MONET

MONET (Millefeuille-like Organization NEural neTwork simulator) is a general-purpose neural circuit simulator developed by Igarashi, one of the authors of this book. The simulator is very fast and scalable for spiking networks composed of LIF models and conductance-based synapses.

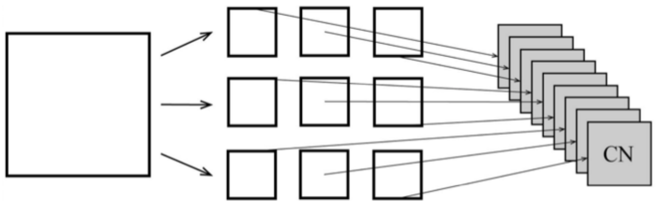


Fig. 17.1 Tile partitioning using MONET. A large network is viewed from above (left), divided into smaller networks of equal size in two dimensions (center), and each network is assigned to an individual computing node to run the simulation (right).

MONET exploits two properties of the structure of the brain's neural circuits.

1. It has a layered structure in the depth direction.
2. About half of the connections are made up of short-range and the other half are made up of long-range.

For property 1, the cerebral cortex is a six-layered structure (section 5.1), and the cerebellum is also a layered structure (section 6.1). The basal ganglia are composed of multiple neuronal nuclei. However, if we consider each neural nucleus as a layer, it can be regarded as a layered structure (Section 7.1. As for property 2, this is correct because neural circuits are not random networks throughout the brain, but are basically tightly coupled within layers and nuclei, and some coupling exists when crossing layers and nuclei. Property 1 shows that MONET is a network with a layered structure. The entire network is divided into tiles perpendicular to the layer direction, like cutting a cake, and the simulation of each divided network is executed on each computational node of the supercomputer (Figure 17.1). From property 2, most of the spike propagation is completed within a node, which reduces the cost of communication. Of course, spike propagation across computation nodes occurs, but the actual inter-node communication is hidden by considering the propagation delay. Furthermore, by using thread parallelism with OpenMP, we have succeeded in demonstrating very good absolute performance and scaling performance. Finally, using 63504 nodes of the K computer, we succeeded in simulating a 1073 cm^2 sheet of cerebral cortex with six layers of about 6 billion neurons and about 25 trillion synapses (Fig. 17.2A, where we simulate the resting state of an animal and the synchronization of neuronal firings appears naturally (Fig. 17.2B).

Human-scale cerebellar simulation

As an example to demonstrate the superior performance of MONET, Hiroshi Yamaura of the University of Electro-Communications (at the time) worked on a large-scale simulation of the cerebellum using the K computer.

As introduced in section 6.1, the cerebellum is also divided into multiple layers, and conveniently, since it is a replication of a microcomplex, it has a sheet-like structure with identical tiles arranged in an orderly fashion. We actually configured the network in such a way and assigned each microcomplex to a node of the K computer by tiling (Fig. 17.3). Each microcomplex can contain about 800,000 granule cells, and we finally succeeded in simulating a cerebellum consisting of about 68 billion neurons using all 82944 nodes of the K computer. The number of 68 billion neurons is almost the same as the number of neurons in the human cerebellum, which means that a human-scale simulation of the cerebellum (in terms of the number of neurons alone) has been achieved.

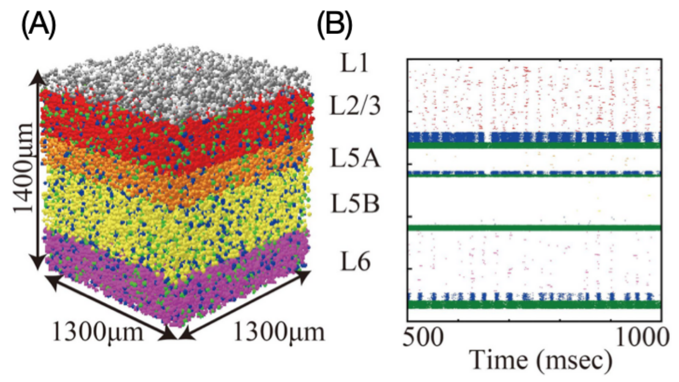


Fig. 17.2 MONET simulation of the cerebral cortex. (A) A single-tiling network. (B) Simulation results.

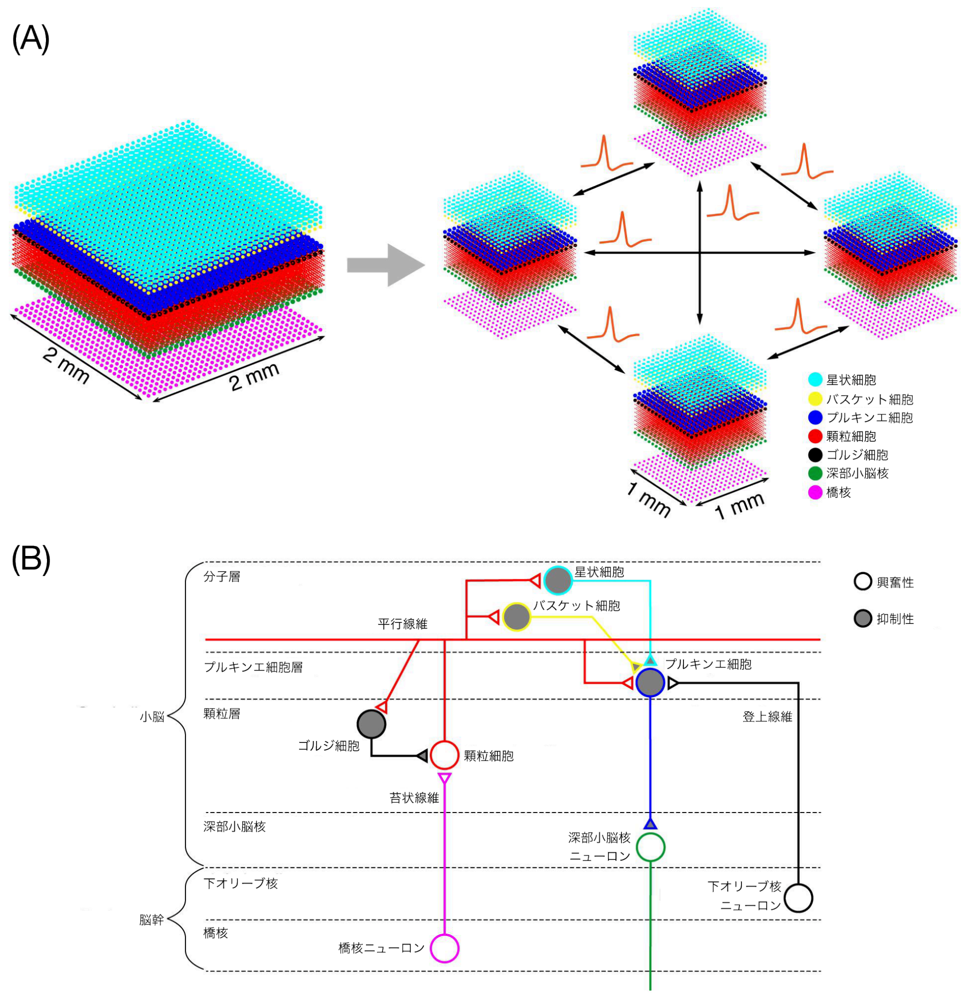


Fig. 17.3 MONET simulation of the cerebellum. (A) Region segmentation. In this example, a $2 \times 2\text{mm}^2$ sheet is divided into $1 \times 1\text{mm}^2$ (B) The structure of the microcomplexes in the divided tiles. (B) Structure of the microcomplex contained in the divided tiles.

17.2.5 Program for Promoting Researches on the Supercomputer Fugaku (2020–)

After the completion of the Post-K Budding Researchers Program, the Program for Promoting Researches on the Supercomputer Fugaku [*15] was launched in FY2020. In this project, as a continuation of the Budding Researchers Project, we are developing a whole-brain simulation for rodents and MONET. We are also working on the transplantation and tuning of the next-generation simulator, NEST 3, to Fugaku, as well as collaboration with the Neurorobotics Platform developed by the Human Brain Project, which is also the Partnering Project of the Human Brain Project. It is also a Partnering Project of the Human Brain Project. As of this writing, we are working on a whole-brain simulation for rodents, which is being led by Dr. Carlos Gutierrez of OIST and Dr. Sun Zhe of RIKEN.

The rodent whole-brain simulation is an attempt to combine models of the cerebral cortex, basal ganglia, and cerebellum, which have been individually researched and developed by RIKEN, OIST, and the University of Electro-Communications, and make them function as a single network as a whole. Each model is relatively faithful to the anatomy and physiology, and the total number of neurons is about 1 million. As of 2021, we have succeeded in reproducing neural activity at rest and in a simple reinforcement learning task (Gutierrez et al., 2019). As of 2021, we have succeeded in reproducing neural activity at rest and in a simple reinforcement learning task (Gutierrez et al., 2019). We are currently transplanting the NEST 3 simulator to the supercomputer ”Fugaku”, and when it is completed, the whole-brain mouse model should start working on Fugaku. Eventually, we plan to scale up to a human-scale whole-brain model using all the nodes of Fugaku.

Igarashi and Yamazaki finally succeeded in simulating a literal human-scale model of the cerebellum, consisting of 96 billion neurons and 57 trillion synapses, by occupying all the nodes of Fugaku and maximizing the performance of the MONET simulator (Igarashi et al., 2021). Needless to say, this is the world’s largest neural circuit simulation as of 2021. Moreover, a biological 1-s simulation can be completed in only 15 seconds. At this point, it is simply a benchmark that we were able to actually run a simulation of that scale, but we will use this model in the future to reproduce and predict the various neural activities of the human brain.

17.3 Future outlook

Now that we have briefly reviewed the recent developments in neural circuit simulation, what will happen in the future? Let’s think about that [*16].

The first thing to consider is whether the performance of supercomputers will continue to increase. If Moore’s Law continues to hold true and supercomputer performance continues to improve along with it, then the question becomes, ”Will supercomputer performance continue to increase? If Moore’s Law continues to hold true and the performance of supercomputers continues to improve along with it, then we will be able to run even larger scale simulations, but if it reaches a plateau at some point, then we will be forced to take appropriate measures. Let us consider each of these scenarios.

17.3.1 If performance continues to improve

Consider 10 years between 2020–2030. Since the process rule of the most advanced semiconductors reached around 10 nm in the mid-2010s, it has become more difficult to shrink the process rule, and the speed of performance improvement has been slowing down. In fact, the Earth Simulator in 2002

[*15] `https://brain-hpc.jp/`

[*16] The following is the personal opinion of the authors as of 2021, and is not a guarantee of the future. It is not a guarantee of the future.

had 40 TFLOPS, and the K computer in 2011 had 11 PFLOPS, a 275-fold improvement, while the Fugaku computer in 2021 will have 488 PFLOPS, only a 40-fold improvement from the K computer. If the performance improvement continues at this pace, it is expected that the performance will increase by 20-40 times in 2030, and the performance of supercomputers may reach about 10-20 E (Exa) FLOPS.

One direction in this scenario is to go to large scale, and to use single-compartment neuron models. One milestone is the development of human-scale whole-brain simulations. Another direction is refinement, where neuron models can be made multicompartmental and replaced by more realistic neurons. Since both of these directions are necessary, the exploration of neural circuits by numerical simulation will continue as long as the performance of computers continues to improve. In addition, the next milestone will be real time simulation, which will be faster than ever before. It will be possible to reconstruct the activity of all neurons with sufficient temporal and spatial resolution, and to reproduce and predict various phenomena.

In addition, in the world of simulation, it is not necessary to target the whole human brain, and it is even possible to introduce more neurons and synapses than humans. Since it is believed that the brain's complex functions are created by the combination of a huge number of neurons, it is very interesting to see whether introducing more neurons than humans will create functions that exceed those of humans [*17] . In particular, is it really possible to create so-called "general-purpose artificial intelligence" or "strong AI"? Maybe we can, maybe we can't, but at times like this, I am always encouraged by Alan Kay's words.

“The best way to predict the future is to invent it” — Alan Kay[*18]

The best way to predict the future is to invent it. I think the point is clear.

17.3.2 When performance stops to improve

As explained in Section 12.1, the performance of supercomputers has been steadily improving according to Moore's Law. As explained in Section 12.1, the performance of supercomputers has been steadily improving according to Moore's Law, but there is debate as to how long this will continue, and some believe that it will soon come to an end. In addition, modern supercomputers increase performance by laying out a large number of computational nodes, but now there is a power problem. For example, "Fugaku" consumes a maximum of about 30 MW/h. Beyond that, it will no longer be possible to use dedicated power plants. Beyond that, a dedicated power plant will no longer be necessary.

Therefore, instead of sacrificing the versatility of computers to specialize in specific calculations, the development of dedicated processors that drastically reduce power consumption is expected to become more popular in the future. Already, the field of deep learning is experiencing a renaissance in the development of dedicated processors that mainly perform inference with low power consumption, and similar efforts are rapidly developing in spiking network simulation (Mead, 1990; Monroe, 2014). This is the last thing I will discuss.

17.3.3 Neuromorphic computing

CPUs today are clocked at a ridiculous speed of several GHz, but neurons can only fire spikes at a rate of 1000 spikes/s at most, which is 106 orders of magnitude less. CPUs consume power mainly by charging and discharging capacitors per clock, so the power is determined roughly in proportion to the number of clocks. On the other hand, neurons also consume power when ion current flows

[*17]However, since the animal with the largest number of neurons in the world is not a human but an African elephant, it is easy to imagine that blindly increasing the number of neurons alone would be difficult.

[*18]`http://ei.cs.vt.edu/~history/GASCH.KAY.HTML`

during firing. Therefore, the operating speed of the neuron, which is lower than 10^6, is one of the factors for low power consumption. In addition, in the current computer architecture, the arithmetic unit and the memory are separated and the memory is read/written by solving the bus, but a lot of power is consumed for memory transfer on the bus. On the other hand, in the case of a neuron, what corresponds to memory is the strength of the synaptic connections, which are integrated with the neuron corresponding to the arithmetic unit. In the case of a neuron, the equivalent of memory is the strength of the synaptic connections, which are integrated with the neuron corresponding to the arithmetic unit, so it is a large-capacity register rather than a memory, and there is no so-called external memory. The equivalent of external memory access is the propagation of spikes between neurons, and the data transferred here is only one bit because it is a spike. Of course, if a huge number of spikes are propagated, memory accesses will increase, but by keeping the firing frequency low, power consumption can be reduced as a result.

With this idea in mind, dedicated digital processors that actually mimic spiking networks, so-called **neuromorphic chips**, are being developed under fierce competition in both the public and private sectors (Merolla et al., 2014; Furber et al., 2014; Friedmann et al., 2017; Davies et al., 2018). The goal is to achieve the same level of advanced information processing as the human brain with the same level of power consumption as the human brain. In 2014, IBM announced TrueNorth, a paper was published in Science, and a very large system is currently being built (DeBole et al., 2019). The inclusion of synaptic plasticity, which was not present in TrueNorth, has greatly increased its value for use. In 2018, Intel announced Loihi, which has synaptic plasticity, something TrueNorth did not have, making it very valuable. e.g., the TrueNorth paper reports very power-efficient computation of the inference part of a deep convolutional network, and the Loihi paper shows similarly very power-efficient computation of MNIST handwriting recognition and L1 norm optimization demos. In October 2021, the second generation Loihi 2 was announced. In the EU's Human Brain Project, SpiNNaker at the University of Manchester and BrainScaleS at the University of Heidelberg are the two major players [*19], and in Japan, processor development and FPGA implementation are underway. In Japan, processor development and FPGA implementation are underway.

In order to actually use a neuromorphic chip, it is of course necessary to implement all the algorithms in spiking neurons. Thus, the implementation and execution of specific computational algorithms as spiking networks is called **neuromorphic computing**. Neuromorphic computation is not neuroscience, but engineering, and is considered to be similar to application development, especially with neurons as the basic unit. If useful algorithms can be developed that can be implemented neuromorphically, they can take advantage of their ultra-low power consumption, making them suitable for use in modern electronics, where power is an important issue. A happy marriage like that between GPUs and deep learning may occur [*20]. In particular, the power-saving nature of neuromorphic chips will be beneficial for edge computing.

On the other hand, attempts to use neuromorphic chips to conduct neuroscience research have also begun (Ananthanarayanan et al., 2009; van Albada et al., 2018). "Why do neurons use spikes as carriers of information?" is a fundamental question in neuroscience. Traditionally, it has been thought that spikes can be propagated on myelinated axons to reach distant neurons without attenuating the signal, but neuromorphic computation may offer a new interpretation in terms of power consumption.

It should be noted that this is not the first time that an attempt has been made to create a dedicated processor to increase the scale, precision, and speed of research, and it has already been done in the field of astronomy since the 1980s (Sugimoto et al., 1990; 伊藤, 2007). In fact, GRAPE, a dedicated processor for astronomy, has influenced the development of neuromorphic chips. In addition, many-body calculations of astronomical objects, which have been done with GRAPE, have been influencing our spiking network simulation research. The development of computational neuroscience will require the development of neuromorphic chips based on the wisdom of our predecessors, as well

[*19] Karlheinz Meier, who led Brain ScaleS, passed away on October 24, 2018.

[*20] For example, uploading your consciousness to a spiking network (渡辺, 2017) is the most killer feeling I have.

as various interactions and collaborations between fields. The future development of computational neuroscience will require the development of neuromorphic chips that draw on the wisdom of our predecessors, as well as various interactions and collaborations among fields.

17.4 Towards establishing “simulation neuroscience”

As a final summary, these efforts have led to the development of a third type of neuroscience, following experimental neuroscience and theoretical/computational neuroscience. It is believed that **simulation neuroscience** will be established (Fan and Markram, 2019).

As discussed in Section 17.3, the potential for the development of neural circuit simulations of the brain continues. In addition, as outlined in Sections 17.1 and 17.2, connectome research is actively progressing, and the blueprints for building models of the brain are becoming more and more detailed. In addition, the number of neurons that can be used to measure brain activity on a large scale is increasing exponentially, and the scale of the brain that can be matched with spiking network simulation is rapidly expanding. It is simulation neuroscience that integrates all of them and advances neuroscience research.

Simulation neuroscience will be a neuroscience that constructs a very precise digital copy of the brain’s neural circuitry on a supercomputer that integrates huge amounts of experimental data and sophisticated mathematical models, and makes itself the target of observation through large-scale simulations. Since the digital copy of the brain can be manipulated artificially at will, it can serve as a tool to elucidate the causal relationships between brain structure and function. The use of this technology is expected to have a variety of ripple effects, including the elucidation of the neural mechanisms of higher brain functions in humans, the elucidation of the mechanisms of various mental and neurological disorders and the establishment of treatment methods, and the development of general-purpose artificial intelligence.

17.5 Column: Why perform large-scale spiking network simulation

The firing activity of each of the vast number of neurons in the brain is difficult to measure or model. When considering the pressure of a gas, for example, it is possible to calculate the pressure from the equation of state of an ideal gas with information on the amount of substance, temperature, and volume, without considering the movement of each gas molecule. In the case of the brain as well, is it not possible to omit the action potential of individual neurons?

It depends on what you want to see and what your goal is. For example, let's say we want to know the cerebral blood flow from a certain neural activity. It is known that there is a correlation between firing frequency and cerebral blood flow on a time scale of s order. In this case, to estimate cerebral blood flow, it is not necessary to know the time of occurrence of action potentials of individual neurons; it is sufficient to know the average firing frequency of the population.

What if we want to know how the brain represents information? In the case of the primary visual cortex, a particular neuron represents the information of a line segment of a particular slope in the visual field by its firing frequency. Therefore, for visual information, we need to look at changes in the firing frequency of individual neurons.

What about the hippocampus? The hippocampus is thought to represent sequence information in terms of the order in which neurons fire spikes within a time frame of 100 ms or so. For example, when three neurons, A, B, and C, fire in sequence, they are thought to represent the sequence ABC (Drieu and Zugaro, 2019). In this case, it is not enough to track the frequency of firings, but it is necessary to deal with the timing of the firings of individual neurons.

After all, what should we look for when examining the information processing mechanisms of the brain?

This is a question that has been on the minds of neuroscientists around the world for many years, and there is still no clear answer. There is no doubt that spikes are involved, but it is not completely clear which neurons represent information in spikes and how. Many possibilities have been proposed, including the frequency of firing, the order of firing, the phase of the firing oscillations, and the strength of the synchronization of the firings.

Therefore, when examining information processing mechanisms in brain models, it is not clear what features of firing should be mimicked in the models, and this has been debated for many years. Therefore, computational neuroscientists believe that brain models of various levels of abstraction, from simple to complex, should be examined in parallel and compared. In fact, models at various levels of description are being used, including mean-field models [*21], single-compartment models, and multi-compartment models.

Now that we have established that we need to examine brain models at various levels of abstraction, there is one ultimate question we should consider. If we try to reproduce human thought in a neural circuit simulation, what should the model mimic?

The brain is a hierarchical information processing system that processes information through interactions within and between levels. The response of the ion channel changes, the membrane potential of individual neurons changes, and the neuronal population cooperates, resulting in the final processing of the brain as a whole. The phenomena occurring at each level of the hierarchy are thought to be responsible for information processing. The various descriptive level models mentioned in the previous section can be roughly mapped to one of the levels. On the other hand, if we try to reproduce human thought by neural circuit simulation, if we reproduce only the features of some hierarchy but not the information processing of other hierarchies, the interaction between the hierarchies will not occur, and we will not be able to reproduce the information processing as a

[*21] Modeling that considers several neurons together as a population, rather than considering each neuron individually. The spikes are also averaged, so the firing frequency is directly output. It can also be approximated in the time direction.

complete system.

Therefore, in order to achieve this ultimate goal, it is necessary to model the vast number of neurons and connections in such a way that they retain all the features that are considered essential for information processing and can interact with each other in a hierarchical manner. To actually realize this, we need computers with enormous performance and brain measurement data, and it will take time to prepare for this. However, the performance of supercomputers and brain measurement technology are rapidly advancing, and based on the speed of progress, it is predicted that this may be realized in 10 to 20 years (今後の HPCI を使った計算科学発展のための検討会, 2017).

Reproducing human thought through human-scale neural circuit simulations sounds like a science fiction dream, but if it is really possible, it has many possibilities in a wide range of fields such as medicine, education, industry, psychology, and philosophy in human society. Another important point is that its realization may provide a clue to solving the problem of the brain and consciousness. In the present day, the mechanism of consciousness and its involvement in the brain's information processing is still not well understood. There are various hypotheses, some suggesting that a specific substance in the brain is involved in the generation of consciousness, others suggesting that it does not matter what type of substance, but that consciousness is generated wherever information processing occurs. Some hypothesize that consciousness is actively involved in the information processing of the brain, while others hypothesize that it is a secondary phenomenon that is not involved in information processing, but is merely a shadow.

We may be able to find some clues to this problem of the brain and consciousness by trying to reproduce human thought through human-scale neural circuit simulation. If consciousness cannot be reproduced by neural circuit simulation, and if consciousness is actively responsible for information processing, then neural circuit simulation alone cannot reproduce the complete information processing of the brain (Dudai and Evers, 2014). In that case, investigating what is missing from the simulation results may help us understand the role of consciousness and create a model of consciousness. On the other hand, if no special treatment is given to consciousness and it can be completely reproduced by neural circuit simulation, then consciousness is a shadow that is not involved in information processing.

Thus, recreating ourselves in a neural circuit simulation is a journey into the depths of consciousness, and recreating the activity of individual neurons is a long way to go.

Chapter 18

Closing

18.1 Afterword

Yamazaki will write a postscript on behalf of the authors.

Ten years have passed since I started research collaboration with Igarshi-san [*1]. In the process, in the summer of 2018, Mr. Ryuichi Maruyama of Morikita Shuppan (at that time) asked me if I would publish a textbook on neural circuit simulation, and I wrote 2/3 of the book with great enthusiasm [*2], but then I ran out of steam [*3]. Then the season came around, and I was forced to stay at home instead of entering the university because of the arrival of the COVID-19, so I decided to take the opportunity to write the rest of the book and proceeded to write it day by day [*4].

Although I told Maruyama-san that I would do everything with spiking neurons, I was fine with that because I had written a lot of text for Parts I and III, but for Part II I had to write most of the code, and I had to go to the trouble of using spikes when it would have been easier to use the rate model. It would have been easy to do it with a normal rate model, but I went to the trouble of using spikes and stepped on a lot of landmines [*5] . It is said that input is dragged down by output, and I learned a lot of things all over again through writing this book. I am very grateful to Mr. Maruyama. I am also grateful to Mr. Ryosuke Miyaji of Morikita Shuppan, who took over the editing duties from Maruyama-san.

Since I started my joint research with Igarashi-san, the Human Brain Project has been launched, the performance of supercomputers has been steadily improved, individuals can easily benefit from parallel computing using GPUs, and new fields such as neuromorphic computing have been launched. As new fields such as neuromorphic computing emerge, I feel that the demand and expectations for spiking networks are steadily increasing. We hope that our knowledge and skills will be useful to you as a pickaxe in the gold rush.

Finally, although the writing of this book began in the fall of 2018, on December 18, 2018, Dr. Masao Ito, a giant in cerebellar research and the man who literally launched neuroscience in our country, passed away. Although Dr. Ito was a neurophysiologist, he showed a remarkable understanding of theoretical research, and I am truly indebted to him. I wish he could have read this book. I pray that he may rest in peace.

[*1] Since 2009 to be exact. By the way, Japanese-style honorific title ”-san” expresses polite, friendly but not too casual emotion, and is gender-free. Why not to use this?

[*2] It was an illusion that I thought I had written 2/3 at the time, and now that I think about it, I've only written about half of it.

[*3] They also said that they were too busy with the national project.

[*4] They also say that they have more free time now that the national project is over.

[*5] However, it was still a good thing that I was able to strongly recognize that rate and spike are different.

About the authors

Tadashi Yamazaki

Associate Professor, Department of Information and Network Engineering, Graduate School of Information Science and Engineering, University of Electro-Communications. He specializes in neuroscience and simulation science. Using supercomputers, he is trying to understand the learning mechanism of the entire brain, especially the cerebellum. His laboratory is NumericalBrain.Org [*6]. He is a member of the Japanese Neural Network Society, the Japanese Neuroscience Society, and the Society for Neuroscience. Doctor of Science.

Jun Igarashi

He is a senior researcher at the RIKEN Center for Computational Science. He specializes in computational neuroscience and computational science. His research interests include the simulation of cortical neural circuits using supercomputers and parallelization methods for neural circuit simulation. He is a member of the Japanese Neural Network Society, the Japanese Neuroscience Society, and the Society for Neuroscience. Doctor of Engineering.

[*6] `https://numericalbrain.org/`

Part V

Appendix

A

A.1 Introduction to numerical simulation for high-school students

TY: This is exclusive in Japanese edition.

A.2 How to use gnuplot

TY: This is exclusive in Japanese edition.

A.3 Source codes

TY: This is exclusive in Japanese edition.

Bibliography

Akar, N. A., Cumming, B., Karakasis, V., Kusters, A., Klijn, W., Peyser, A., and Yates, S. (2019). Arbor – a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures. 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).

Albus, J. S. (1971). A theory of cerebellar function. Math Biosci, 10:25–61.

Amunts, K., Ebell, C., Muller, J., Telefont, M., Knoll, A., and Lippert, T. (2016). The human brain project: Creating a european research infrastructure to decode the human brain. Neuron, 92(3):574–581.

Ananthanarayanan, R., Esser, S. K., Simon, H. D., and Modha, D. S. (2009). The cat is out of the bag: Cortical simulations with 10^9 neurons, 10^{13} synapses. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA. Association for Computing Machinery.

Anwar, H., Roome, C. J., Nedelescu, H., Chen, W., Kuhn, B., and De Schutter, E. (2014). Dendritic diameters affect the spatial variability of intracellular calcium dynamics in computer models. Frontiers in Cellular Neuroscience, 8:168.

Arkhipov, A. et al. (2018). Visual physiology of the layer 4 cortical circuit in silico. PLOS Computational Biology, 14(11):1–47.

Ascoli, G. A. (2006). Mobilizing the base of neuroscience data: the case of neuronal morphologies. Nature Reviews Neuroscience, 7(4):318–324.

Azevedo, F. A. et al. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. Journal of Comparative Neurology, 513:532–541.

Bliss, T. V. and Lomo, T. (1973). Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. J Physiol, 232(2):331–356.

Bliss, T. V. P. and Collingridge, G. L. (1993). A synaptic model of memory: long-term potentiation in the hippocampus. Nature, 361:31—39.

Bosman, C. A. and Aboitiz, F. (2015). Functional contraints in the evolution of brain circuits. Frontiers in Neuroscience, 9:303.

Branco, T., Clark, B. A., and Häusser, M. (2010). Dendritic discrimination of temporal input sequences in cortical neurons. Science, 329(5999):1671–1675.

Brette, R. et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. J Comput Neurosci, 23(3):349–398.

Brian Simulator (2017). Example: Cuba. `http:// brian2.readthe docs.io/en/stable/examples/CUBA.html` (最終アクセス 2017 年 10 月 17 日).

Caporale, N. and Dan, Y. (2008). Spike timing-dependent plasticity: A hebbian learning rule. Annual Review of Neuroscience, 31(1):25–46.

Carnevale, N. T. and Hines, M. L. (2006). The NEURON Book. Cambridge University Press.

Chandra, R. et al. (2000). Parallel Programming in OpenMP. Morgan Kaufmann.

Christian, K. M. and Thompson, R. F. (2003). Neural substrates of eyeblink conditioning: Acquisition and retention. Learn Mem, 11:427–455.

Connor, J. and Stevens, C. (1971). Prediction of repetitive firing behaviour from voltage clamp data on an isolated neurone soma. J Physiol, 213(1):31–53.

Connor, J. A., Walter, D., and McKown, R. (1977). Neural repetitive firing: modifications of the hodgkin-huxley axon suggested by experimental results from crustacean axons. Biophys J, 18(1):81–102.

Davies, M. et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. IEEE Micro, 38(1):82–99.

Dayan, P. and Abbott, L. F. (2005). Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems. The MIT Press.

DeBole, M. V., Taba, B., Amir, A., Akopyan, F., Andreopoulos, A., Risk, W. P., Kusnitz, J., Ortega Otero, C., Nayak, T. K., Appuswamy, R., Carlson, P. J., Cassidy, A. S., Datta, P., Esser, S. K., Garreau, G. J., Holland, K. L., Lekuch, S., Mastro, M., McKinstry, J., di Nolfo, C., Paulovicks, B., Sawada, J., Schleupen, K., Shaw, B. G., Klamo, J. L., Flickner, M. D., Arthur, J. V., and Modha, D. S. (2019). Truenorth: Accelerating from zero to 64 million neurons in 10 years. Computer, 52(5):20–29.

Doya, K. (1999). What are the computations of the cerebellum, the basal ganglia and the cerebral cortex? Neural Networks, 12:961–974.

Doya, K. (2000a). Complementary roles of basal ganglia and cerebellum in learning and motor control. Curr Opin Neurobiol, 10(6):732–739.

Doya, K. (2000b). Reinforcement learning in continuous time and space. Neural Computation, 12:219–245.

Drieu, C. and Zugaro, M. (2019). Hippocampal sequences during exploration: Mechanisms and functions. Frontiers in Cellular Neuroscience, 13:232.

Dudai, Y. and Evers, K. (2014). To simulate or not to simulate: What are the questions? Neuron, 84:254–261.

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, Y., and Rasmussen, D. (2012). A large-scale model of the functioning brain. Science, 338(6111):1202–1205.

Erisir, A., Lau, D., Rudy, B., and Leonard, C. (1999). Function of specific K^+ channels in sustained high-frequency firing of fast-spiking neocortical interneurons. J Neurophysiol, 82(5):2476–2489.

Fan, X. and Markram, H. (2019). A brief history of simulation neuroscience. Frontiers in Neuroinformatics, 13:32.

Frémaux, N., Sprekeler, H., and Gerstner, W. (2013). Reinforcement learning using a continuous time actor-critic framework with spiking neurons. PLOS Computational Biology, 9(4):1–21.

Friedmann, S., Schemmel, J., Grübl, A., Hartel, A., Hock, M., and Meier, K. (2017). Demonstrating hybrid learning in a flexible neuromorphic hardware system. IEEE Trans Biomed Circ Sys, 11(1):128–142.

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The spinnaker project. Proc IEEE, 102(5):652–665.

Gerstner, W. and Kistler, W. M. (2002). Spiking Neuron Models. Cambridge University Press.

Gerstner, W., Kistler, W. M., Naud, R., and Paninski, L. (2014). Neuronal Dynamics: From Single Neurons To Networks and Models of Cognition. Cambridge University Press.

Gewaltig, M.-O. and Diesmann, M. (2007). Nest (neural simulation tool). Scholarpedia, 2(4):1430.

Gidon, A. et al. (2020). Dendritic action potentials and computation in human layer 2/3 cortical neurons. Science, 367(6473):83–87.

Glickstein, M. (1994). Cerebellar agenesis. Brain, 117:1209–1212.

Goodman, D. and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. Frontiers in Neuroinformatics, 2:5.

Gosui, M. and Yamazaki, T. (2016). Real-world-time simulation of memory consolidation in a large-scale cerebellar model. Front Neuroanat, 10:1–10.

Grillner, S., Wallen, P., Brodin, L., , and Lansner, A. (1991). Neuronal network generating locomotor behavior in lamprey: circuitry, transmitters, membrane properties and simulations. Annual Review of Neuroscience, 14:169–199.

Gropp, W., Lusk, E., and Skjellum, A. (1999). Using MPI: Portable Parallel Programming with the

Message Passing Interface. MIT Press, 2nd edition.

Gutierrez, C. et al. (2019). A whole-brain spiking neural network model linking basal ganglia, cerebellum, cortex and thalamus. In 28th Annual Computational Neuroscience Meeting: CNS*2019, volume 20 (Suppl 1), page P73. BMC Neuroscience.

Harris, M. (2007). Optimizing parallel reduction in CUDA. NVIDIA. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

Hazan, H., Saunders, D. J., Khan, H., Patel, D., Sanghavi, D. T., Siegelmann, H. T., and Kozma, R. (2018). Bindsnet: A machine learning-oriented spiking neural networks library in python. Frontiers in Neuroinformatics, 12:89.

Hebb, D. O. (1949). The organization of behavior; a neuropsychological theory. Wiley.

Helias, M. et al. (2012). Supercomputers ready for use as discovery machines for neuroscience. Frontiers in Neuroinformatics, 6:26.

Hennessy, J. L. and Patterson, D. A. (2017). Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 6th edition.

Hensch, K. T. and Stryker, M. P. (2004). Columnar architecture sculpted by gaba circuits in developing cat visual cortex. Science, 303(5664):1678–1681.

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. Jounal of Physiology, 117(4):500–544.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. Proc Natl Acad Sci USA, 79:2554–2558.

Igarashi, J. et al. (2021). Toward simulation of a human-scale cortico-cerebello-thalamic circuit using supercomputer Fugaku. In Proceedings of The 44th Annual Meeting of the Japan Neuroscience Society.

Igarashi, J., Shouno, O., Fukai, T., and Tsujino, H. (2011). Real-time simulation of a spiking neural network model of the basal ganglia circuitry using general purpose computing on graphics processing units. Neural Networks, 24(9):950–960. Multi-Scale, Multi-Modal Neural Modeling and Simulation.

Igarashi, J., Yamaura, H., and Yamazaki, T. (2019). Large-scale simulation of a layered cortical sheet of spiking network model using a tile partitioning method. Frontiers in Neuroinformatics, 13:71.

Ito, M. (1970). Neurophysiological aspects of the cerebellar motor control system. International Journal of Neurology, 7(2):162–176.

Ito, M. (2012). The cerebellum: Brain for the implicit self. FT Press.

Ito, M., Sakurai, M., and Tongroach, P. (1982). Climbing fibre induced depression of both mossy fibre responsiveness and glutamate sensitivity of cerebellar purkinje cells. J Physiol, 324:113–134.

Izhikevich, E. M. (2003). Simple model of spiking neurons. IEEE Transactions of Neural Networks, 14:1569–1572.

Izhikevich, E. M. (2010). Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting. MIT Press.

Izhikevich, E. M. and Edelman, G. M. (2008). Large-scale model of mammalian thalamocortical systems. Proceedings of the National Academy of Sciences, 105(9):3593–3598.

Jankovic, J. (2008). Parkinson’ s disease: clinical features and diagnosis. Journal of Neurology, Neurosurgery & Psychiatry, 79:368–376.

Kandel, E. R., Siegelbaum, S. A., Mack, S. H., and Koester, J., editors (2021). Principles of neural science. McGraw-Hill Medical, 6th edition.

Koch, C. and Segev, I., editors (1998). Methods in Neuronal Modeling: From Ions to Networks. Bradford Book.

Kohonen, T. (2001). Self-Organizing Maps. Springer, 3rd edition.

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., and Schürmann, F. (2019). Coreneuron : An optimized compute engine for the neuron simulator. Frontiers in Neuroinformatics, 13:63.

Kunkel, S., Schmidt, M., Eppler, J. M., Plesser, H. E., Masumoto, G., Igarashi, J., Ishii, S., Fukai, T., Morrison, A., Diesmann, M., and Helias, M. (2014). Spiking network simulation code for petascale computers. Frontiers in Neuroinformatics, 8:78.

Markram, H. (2005). The blue brain project. Nature Reviews Neuroscience, 7(2):153–160.

Markram, H. et al. (2015). Reconstruction and simulation of neocortical microcircuitry. Cell, 163(2):456–492.

Marr, D. (1969). A theory of cerebellar cortex. J Physiol (Lond), 202:437–470.

Martone, M. E. et al. (2003). The cell-centered database: a database for multiscale structural and protein localization data from light and electron microscopy. Neuroinformatics, 1(4):379–395.

Mascagni, M. V. and Sherman, A. S. (1998). Numerical methods for neuronal modeling. In Koch, C. and Segev, I., editors, Methods in Neuronal Modeling: From Ions to Networks, pages 569–606. The MIT Press, 2nd edition.

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul., 8(1):3–30.

Matsuoka, K. (1985). Sustained oscillations generated by mutually inhibiting neurons with adaptation. Biological Cybernetics, 52:367–376.

Mauk, M. D. and Donegan, N. H. (1997). A model of Pavlovian eyelid conditioning based on the synaptic organization of the cerebellum. Learn Mem, 3:130–158.

Mead, C. (1990). Neuromorphic electronic systems. Proceedings of the IEEE, 78(10):1629–1636.

Merolla, P. A. et al. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. Science, 345(6197):668–673.

Miikkulainen, R., Bednar, J. A., Choe, Y., and Sirosh, J. (2005). Computational Maps in the Visual Cortex. Springer.

Miller, K., Keller, J., and Stryker, M. (1989). Ocular dominance column development: analysis and simulation. Science, 245(4918):605–615.

Miller, K. D. (1994). A model for the development of simple cell receptive fields and the ordered arrangement of orientation columns through activity-dependent competition between on- and off-center inputs. Journal of Neuroscience, 14(1):409–441.

Miyashita, M. and Tanaka, S. (1992). A mathematical model for the self-organization of orientation columns in visual cortex. NeuroReport, 3:625–640.

Moldwin, T. and Segev, I. (2020). Perceptron learning and classification in a modeled cortical pyramidal cell. Frontiers in Computational Neuroscience, 14:33.

Monroe, D. (2014). Neuromorphic computing gets ready for the (really) big time. Commun. ACM, 57(6):13–15.

Nakano, K. (1972). Associatron — a model of associative memory. IEEE Transactions on Systems, Man, and Cybernetics, SMC-2(3):380–388.

Okano, H., Erika, S., Yamamori, T., Iriki, A., Shimogori, T., Yamaguchi, Y., Kasai, K., and Miyawaki, A. (2016). Brain/MINDS: A japanese national brain project for marmoset neuroscience. Neuron, 92(3):582–590.

Rall, W. (1964). Theoretical significance of dendritic trees for neuronal input-output relations. In Reiss, R. F., editor, Neural Theory and Modeling. Stanford Univ Press.

Ramos, K. M. et al. (2019). The NIH BRAIN initiative: Integrating neuroethics and neuroscience. Neuron, 101(3):394–398.

Rieke, F., Warland, D., de Ruyter von Steveninck, R., and Bialek, W. (1997). Spikes: Exploring the Neural Code. MIT Press.

Rosenblatt, M. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychol Rev, 65:386–408.

Rotter, S. and Diesmann, M. (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. Biological Cybernetics, 81(5-6):381–402.

Samejima, K., Ueda, Y., Doya, K., and Kimura, M. (2005). Representation of action-specific reward values in the striatum. Science, 310:1337–1340.

Schultz, W. (1998). Predictive reward signal of dopamine neurons. Journal of Neurophysiology, 80:1–27.

Schultz, W., Dayan, P., and Montague, P. R. (1997). A neural substrate of prediction and reward. Science, 275(5306):1593–1599.

Schwiening, C. J. (2012). A brief historical perspective: Hodgkin and Huxley. Journal of Physiology, 590(11):2571–2575.

Segev, I., Rinzel, J., and Shepherd, G. M., editors (1994). The Theoretical Foundation of Dendritic Function. MIT Press.

Shatz, C. J. and Stryker, M. P. (1978). Ocular dominance in layer iv of the cat's visual cortex and the effects of monocular deprivation. Journal of Physiology, 281:267–283.

Shimazaki, H. and Shinomoto, S. (2007). A method for selecting the bin size of a time histogram. Neural Computation, 19(6):1503–1527.

Shouno, O., Tachibana, Y., Nambu, A., and Doya, K. (2017). Computational model of recurrent subthalamo-pallidal circuit for generation of parkinsonian oscillations. Frontiers in Neuroanatomy, 11:21.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529:484–489.

Slotnick, D. L. (1982). The conception and development of parallel processors: A personal memoir. Annals of the History of Computing, 4(1):20–30.

Srinivasan, R. and Chiel, H. J. (1993). Fast calculation of synaptic conductances. Neural Computation, 5:200–204.

Sugimoto, D., Chikada, Y., Makino, J., Ito, T., Ebisuzaki, T., and Umemura, M. (1990). A special-purpose computer for gravitational many-body problems. Nature, 345:33–35.

Sutton, R. S. and Barto, A. G. (2018). Reinforcement Learning: An Introduction. The MIT Press, 2nd edition.

Swindale, N. V. (1996). The development of topography in the visual cortex: a review of models. Network: Comput. Neural Syst., 7:161–247.

Taga, G., Yamaguchi, Y., and Shimizu, H. (1991). Self-organized control of bipedal locomotion by neural oscillators in unpredictable environment. Biol. Cybern., 65:147–159.

Tanabe, M., Gähwiler, B. H., and Gerber, U. (1998). L-type Ca^{2+} channels mediate the slow Ca^{2+}-dependent afterhyperpolarization current in rat ca3 pyramidal cells in vitro. Journal of Neurophysiology, 80(5):2268–2273.

Tanaka, S. (1990). Theory of self-organization of cortical maps: Mathematical framework. Neural Networks, 3(6):625–640.

Trappenberg, T. P. (2010). Fundamentals of Computational Neuroscience. Oxford University Press, 2nd edition.

Traub, R. D. and Wong, R. K. S. (1991). A model of CA3 hippocampal pyramidal neuron incorporating voltage-clamp data on intrinsic conductances. Journal of Neurophysiology, 66(2):635–650.

Trevelyan, A. J., Sussillo, D., Watson, B. O., and Yuste, R. (2006). Modular propagation of epileptiform activity: evidence for an inhibitory veto in neocortex. J Neurosci, 26(48):12447–12455.

Treves, A. (1993). Mean-field analysis of neuronal spike dynamics. Network: Computation in Neural Systems, 4:259–284.

Tsai, P. T. et al. (2012). Autistic-like behaviour and cerebellar dysfunction in Purkinje cell Tsc1 mutant mice. Nature, 488:647–651.

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., Lester, D. R., Diesmann, M., and Furber, S. B. (2018). Performance comparison of the digital neuromorphic hardware spinnaker and the neural network simulation software nest for a full-scale cortical microcircuit model. Frontiers in Neuroscience, 12:291.

Yamaura, H., Igarashi, J., and Yamazaki, T. (2020). Simulation of a human-scale cerebellar network model on the k computer. Frontiers in Neuroinformatics, 14:16.

Yamazaki, T. (2021). Evolution of the Marr-Albus-Ito model. In Mizusawa, H. and Kakei, S., editors, Cerebellum as a CNS Hub, page In Press. Springer.

Yamazaki, T. and Igarashi, J. (2013). Realtime cerebellum: A large-scale spiking network model of the cerebellum that runs in realtime using a graphics processing unit. Neural Networks, 47(11):103–111.

Yamazaki, T., Igarashi, J., Makino, J., and Ebisuzaki, T. (2019). Real-time simulation of a cat-scale artificial cerebellum on PEZY-SC processors. Int J High Perf Comp App, 33(1):155–168.

Yamazaki, T., Igarashi, J., and Yamaura, H. (2021). Human-scale brain simulation via supercomputer: a case study on the cerebellum. Neuroscience, 462:235–246.

Yamazaki, T. and Lennon, W. (2019). Revisiting a theory of cerebellar cortex. Neurosci Res, 148(2019):1–8.

Yamazaki, T. and Nagao, S. (2012). A computational mechanism for unified gain and timing control in the cerebellum. PLoS ONE, 7(3):e33319.

Yamazaki, T. and Tanaka, S. (2005). Neural modeling of an internal clock. Neural Computation, 17:1032–1058.

Yamazaki, T. and Tanaka, S. (2007). A spiking network model for passage-of-time representation in the cerebellum. Eur J Neurosci, 26:2279–2292.

Yavuz, E., Turner, J., and Nowotny, T. (2016). Genn: a code generation framework for accelerated brain simulations. Scientific Reports, 6:18854.

ニコリ, editor (2006). 決定版数独 (パズル通信ニコリ別冊). ニコリ.

伊藤, 智. (2007). スーパーコンピューターを 20 万円で創る. 集英社.

皆本, 晃. (2005). C 言語による数値計算入門 — 解法・アルゴリズム・プログラム. サイエンス社.

宮本, 大., 加沢, 知., and 神崎, 亮. (2015). 昆虫嗅覚系全脳シミュレーションに向けて — スーパコンピュータによる大規模脳シミュレーションの 現在とその展望 —. 人工知能, 30(5):630–638.

五十嵐, 潤., 庄野, 修., Moren, J., 吉本, 潤., and 銅谷, 賢. (2015). パーキンソン病の運動症状の発生機構解明にむけた大脳基底核–視床–大脳皮質回路のスパイキングニューロンモデルの開発. 日本神経回路学会誌, 22(3):103–111.

今後の HPCI を使った計算科学発展のための検討会, editor (2017). 計算科学ロードマップ 2017. `https://cs-forum.github.io/kentoukai/roadmap-2017/`.

神崎, 亮., editor (2018). 昆虫の脳をつくる — 君のパソコンに脳をつくってみよう —. 朝倉書店.

渡辺, 正. (2017). 脳の意識 機械の意識 - 脳神経科学の挑戦. 中公新書.

片桐, 孝. (2013). スパコンプログラミング入門: 並列処理と MPI の学習. 東京大学出版会.

Index