

A Discrete-Event Neural Network Simulator for General Neuron Models

T. Makino

Department of Complexity Science and Engineering, Graduate School of Frontier Science, Tokyo University, Tokyo, Japan

Efficient simulation techniques for a discrete-event pulsed neural network simulator are developed. In a discrete-event simulation framework, simulation of complex neural behaviours, such as phase precession and phase arbitration, demands the prediction of delayed firing times. The new technique, the incremental partitioning method, uses linear envelopes of the state variable of a neuron to partition the simulated time so that the delayed-firing time is reliably calculated by applying the bisection-combined Newton-Raphson method to every partition. The quick filtering technique is also proposed for reducing calculation cost of linear envelopes. The simulator developed, PUNNETS, has achieved efficiency and precision, but is still capable of simulating a complex behaviour of large-scale neural network models.

Keywords: Discrete-event simulation; Event-driven simulation; Incremental partitioning method; Neural network simulator; PUNNETS; Pulsed neural network

1. Introduction

The importance of time in a neural network simulation is increasing. Emerging research areas, such as the simulation of memory and context handling in a neural network, are requiring the simulation of temporal transitions of the network. Recent studies pointed out that temporal coincidence of pulses have various roles in the brain, including binding enco-

ding [1,2] and functional connectivity [3]. A high-precision and efficient simulator for pulsed neural networks is demanded for studying the temporal behaviour of the brain.

Most existing simulators are based on a discrete-time simulation framework (also known as synchronous simulation) [4,5]. Although this framework is easy to develop, it inevitably requires a large amount of computation to increase temporal precision. If the temporal precision is reduced to achieve efficiency, pulse timings are restricted and the expressive power of temporal coding decreases.

It is widely known that a discrete-event simulation framework, also called event-driven simulation, can simulate a neural network with high temporal precision. Studies on discrete-event neural network simulation were pioneered by Watts [6], and application to a larger network has been investigated by various researchers [7,8]. However, the neuron model in the existing simulators is restricted to a rather simple class, in which the future transition of the neuron is easily predictable. Techniques to simulate a more complex class of neuron models are thus being demanded by advanced simulation tasks, such as the simulation of the short-term memory model of hippocampus.

It is known that most of the demanded neuron models can be described by the Spike-Response model, whose state is described as a summation of presynaptic pulse-response functions, a self-spike response function and an external input function. This model includes a large class of neurons, such as leaky integrate-and-fire neurons [9]. However, its high expressive power makes it difficult to predict the future behaviour of a neuron, especially to detect the nearest threshold-crossing point that corresponds to the next firing time.

Correspondence and offprint requests to: T. Makino, Department of Complexity Science and Engineering, Graduate School of Frontier Science, Tokyo University, Hongo 7-3-1, Bunkyo-ku, 113-0033 Tokyo, Japan. Email: mak@sat.t.u-tokyo.ac.jp

In response to the above-described situation, we developed a *second-order incremental partitioning method*, which is a general solver to detect the nearest threshold-crossing point by using linear envelopes of a function and its derivatives. The linear envelopes can be defined for any C_1 -class continuous function; even when the function has discontinuities, we can partition the function into continuous parts. Moreover, since linear envelopes of various functions can be summed, this method is easily applicable to a neuron model with any functions splittable into finite ranges of second-order differentiable functions, including the Spike-Response model of a neuron.

We also devised a filtering technique for reducing the cost of the partitioning method. Since the partitioning method is based on prediction of the future, every arrival of a pulse causes recalculation of the prediction, which degrades the efficiency. Our technique, *quick filtering by maximum gradient checking*, effectively reduces the number of predictions by concerning the next-known-pulse arrival at a neuron.

2. Discrete-Event Neural Network Simulation

Numerical simulation of neural networks is commonly based on a discrete-time simulation framework. In discrete-time simulation, the temporal transition of neural states are represented in a form of associated differential equations. The values of state variables are then updated synchronously for each time step Δt , using a finite integration method such as Euler or Runge-Kutta. Δt gives temporal resolution of the simulation in a sense that the simulator cannot reproduce dynamics in a time span less than Δt . Since the simulation cost is inversely proportional to Δt , a coarse temporal resolution must be used for large-scale network simulations.

For the simulation of pulsed neural networks, the discrete-time simulation framework is not suitable. To simulate the temporal correlation of pulses, Δt must be significantly less than the correlating pulses, so the performance of the simulation degrades drastically. In addition, when the framework is applied to pulsed neural networks, most of the calculation is a deterministic update of neuron states. In a pulsed neural network, neurons intercommunicate with pulses. The transition of a neuron state between receiving pulses is deterministic. In the case of a fine-grained time step, most of the synchronous updates in discrete-time simulation concern deterministic evolution of neuron states. If this evolution

were properly calculated, such synchronous updates could be reduced.

Elaborating this idea, we obtain a different framework of simulation, which is called a discrete-event simulation framework. An arrival of a pulse to a neuron is regarded as an event; the state of the neuron is calculated only at the time an event occurs. This process may cause the neuron to fire, which causes new pulses to be sent, each of which turns into another event. This framework is called discrete-event because it cannot simulate continuous interaction of neurons; that is, it can only simulate a discrete sequence of events. However, it is a suitable framework for pulsed neural networks, in which every interaction of neurons is a discrete pulse.

2.1. Discrete-Event Simulation of a Neural Network

Figure 1 sketches a discrete-event simulation process with a simple integrate-and-fire neuron model. The simulator keeps information of each neuron as a pair consisting of the last simulation time and the value of the state variable at that time, which are denoted in the figure as ‘Last’ and ‘Sig’, respectively. A scheduling queue keeps pending events in the order of arrival time.

The simulation process consists of the repeated deliveries of the earliest pending event in the scheduling queue to the neuron. In the figure, the event arriving at neuron A at time 5.0 is the earliest pending event; thus it is delivered to neuron A. Then the state of the neuron is updated to the time of the event. In this case, the last simulation of neuron A was at time 4.0, and the state variable at that time was 0.7. As the event arrived at time $t = 5.0$, the state of neuron A is updated to time 5.0: Last becomes 5.0, and Sig is updated to 0.4, i.e. the decayed value at $t = 5.0$. Note that, in this update process, other neurons such as neuron B are kept unchanged. The calculation of the state of A presumes no other pulse arrives at A before that time, although the state of neuron B, which may send pulses to A, is left uncalculated from $t = 3.3$. This is because we know that neuron B never fires unless it receives an external pulse, and pulses for B are absent between the last calculation of the state of B ($t = 3.3$) and the calculation of the state of A ($t = 5.0$). The absence of the pulses is ensured by the scheduling queue, which stores events and serves them in the order of arrival time. In this way, discrete-event simulation keeps the whole network consistent while minimising the neuron states to be updated.

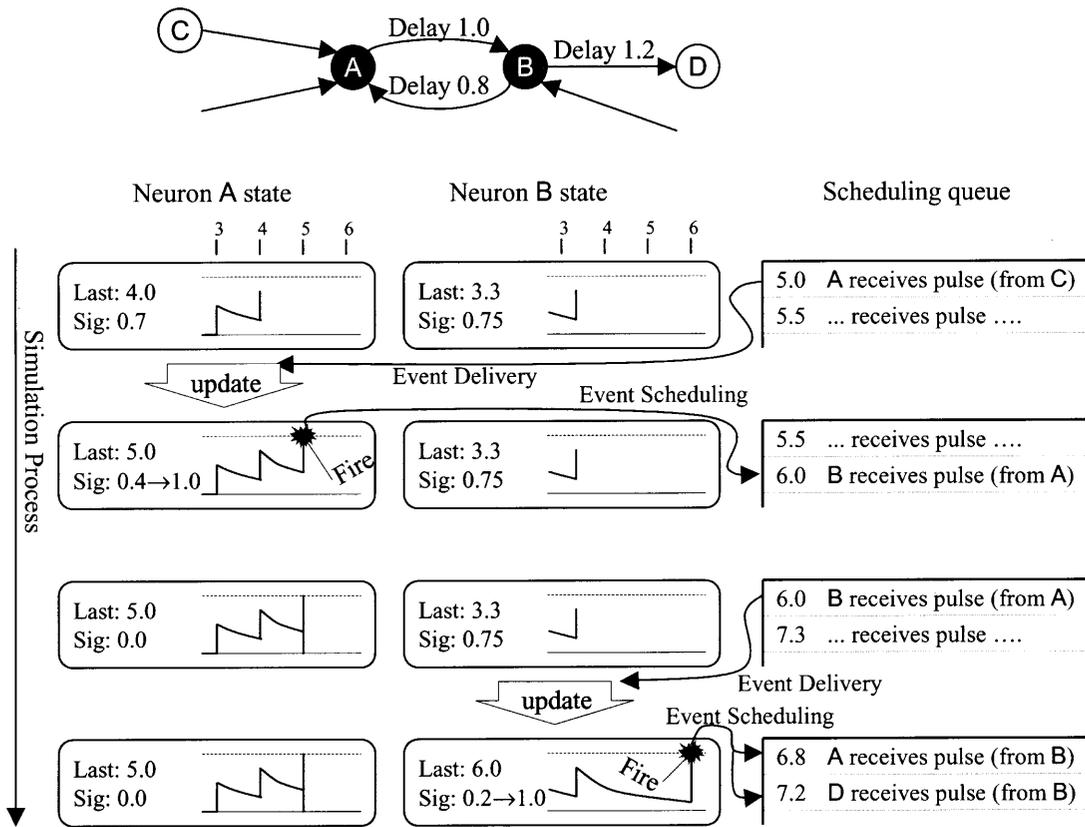


Fig. 1. Discrete event simulation model.

Thereafter, the effect of the pulse is added to neuron A, which causes A to fire at time 5.0. As a result, A sends a pulse to neuron B, with a delay of time 1.0. Thus, an event of pulse arrival at B is scheduled at time 6.0. When the event comes to the top of the queue, it is delivered to neuron B, and at that time the state of B is updated. If the event caused firing, then another set of new events is scheduled. In this way, the repeated deliveries constitutes the simulation.

As described above, in a discrete-event simulation framework, the update process of states no longer relies on synchronous processing of neurons in Δt steps, but on calculation based on event arrivals. This advantage makes it easy to achieve high temporal precision efficiently with pulsed neural networks.

2.2. Delayed Firing

One remaining problem is the handling of *delayed firings*. In some cases, the effect of an event on a neuron is not instant. In the upper part of Fig. 2, the pulse itself does not cause immediate firing, but causes the neuron to fire at a later time. The hand-

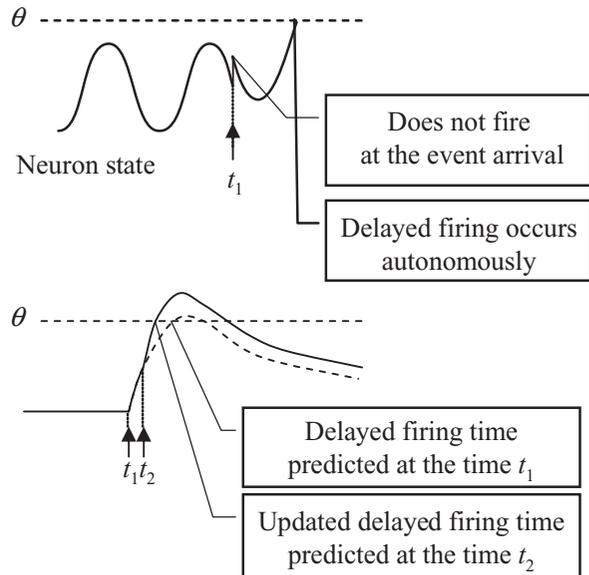


Fig. 2. Delayed firing of a neuron. The upper part shows a simple sine function with an immediate response for the pulse at time t_1 . The lower part shows spike-response functions for the pulses at t_1 and t_2 . In the latter case, the first prediction of the firing time at t_1 is changed by another pulse arrival at t_2 .

ling of such a firing, which we call delayed firing, poses a problem for discrete-event simulation. Namely, since the neuron state is not calculated until the arrival of the next event, the delayed firing is ‘ignored’ until the arrival of the next event. If the pulses produced by the delayed firing are not simulated in order of arrival time, the causality of the simulation system is violated.

In a general neuron model such as the Spike-Response model, delayed firing is not a special case. If a response function such as the one in the lower part of Fig. 2 is used, a firing is always delayed from the last pulse arrival. Moreover, a superposed response function from a later arrival of another pulse causes the change in the delayed firing time. Such a change poses more difficulty for the simulation.

To avoid this problem, delayed firing has to be scheduled in the pending event queue, which requires prediction of the precise timing of the delayed firing when the previous event is processed. This firing prediction is undoubtedly the key to precise simulation of pulsed neural networks. However, it is difficult to predict firing for a complex neuron model such as the Spike-Response model, as described in the next section.

2.3. Difficulty of Delayed Firing Prediction

Simulating complex neuron models, including the Spike-Response model of pulsed neural networks, are demanded in neural modeling of human memory and high-level information tasks using human memory [2]. Such a neuron model is described by a summation of a number of functions of time t , including exponential and trigonometric functions. However, it is difficult to predict the time of delayed firing for such a neuron.

The difficulty is caused by the mathematical complexity involved in finding the time of delayed firing. Even if we can give a functional expression to the state variable $u_i(t)$, it is different from finding roots of the equation $u_i(t) = \theta$, which gives the firing time. Analytical methods for finding a root are restricted to simple functions, such as a linear function and a simple exponential function. In general, we cannot analytically find the roots for an equation that is a summation of several exponential and trigonometric functions; it is more difficult than finding roots of higher-order polynomial equations.

However, we can solve such an equation numerically. The Newton-Raphson method is one of the best-known and most powerful methods to give a numerical solution to an equation. Basically, in solv-

ing an equation $f(x) = 0$, the method repeatedly moves variable x to a crossing point of the x -axis and the tangent line of $f(x)$ at point x until x converges on a root.

Although the simple application of the Newton-Raphson method sometimes fails to find a root, it is known that the Newton-Raphson method combined with the bisection method can safely find a root if we enclose the root in a range [10]. Here, *enclosing* means finding a range (x_1, x_2) for a function $f(x)$ in which the values $f(x_1)$ and $f(x_2)$ have the opposite signs; at least one root exists in the range because the function is continuous. Since the bisection-combined Newton-Raphson method is applicable to any differentiable function, it is suitable to find a root of $u_i(t) = \theta$, where u_i is a sum of differentiable functions.

Nevertheless, the method is still incomplete, i.e. it cannot predict the delayed firing time. Figure 3 illustrates a situation comprising a sum of a linear function and a sine function. A prediction algorithm of the delayed firing time should correctly find the first point beyond the threshold, which is time t_0 in the figure. However, we cannot control which root is calculated by the Newton-Raphson method; namely, the method may converge to any root, such as t_S , the second crossing of the threshold.

For accurate simulation we have to guarantee that the solver finds the first threshold-crossing point. However, it is difficult to distinguish it from *false crossing*, such as t_F , where the state variable approaches but does not go beyond the threshold. When the solver finds t_0 as a root, how can we guarantee that all previous approaches to the threshold are all false crossings? This is a difficult question for a Spike-Response-model neuron,

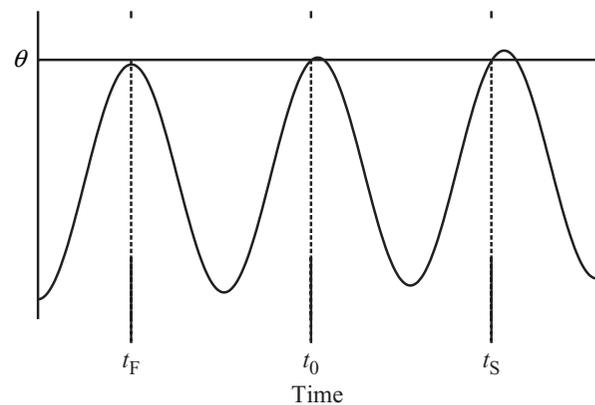


Fig. 3. Difficulty in finding the firing. We want to find t_0 , the first threshold-crossing time. However, it is difficult for a solver to distinguish it from t_F , where the state variable approaches but does not go beyond the threshold, and from t_S , the second threshold-crossing time.

because many exponential and trigonometric functions are superimposed to form its state function. The next section describes our method to solve this problem.

3. Incremental Partitioning Method

3.1 Overview of the Incremental Partitioning Method

Partitioning is a simple idea to solve the difficulty concerning the Newton-Raphson method. We divide the function into *partitions*, each of which has at most one threshold-crossing point. After that, we check each partition to see whether it has a crossing point, and apply the Newton-Raphson method for the first partition containing the crossing point. This method can find the first crossing point, i.e. the time of the delayed firing, without mistakenly finding the second and later crossing points.

In the simulation, all partitions do not need to be solved at once; they can be calculated and solved one partition at a time. Figure 4 illustrates this process. When a partition containing the current simulation time t is solved but no crossing in the partition is found, calculating and solving the next partition can be postponed until the simulation time reaches the end of the partition. The postponement is done by scheduling the solution of the next partition as an event. We call this method *incremental partitioning*.

This method is suitable for discrete-event neural network simulation for the following reasons. First, scheduling of the next partitioning can be implemented in a consistent way with scheduling of

other events, such as firing and pulse arrival. Secondly, it uses more computing power for the near future; since a new arrival of pulses easily changes the state of the neuron, it is often redundant to predict firings in the distant future.

The remaining problem is providing an algorithm for partitioning. If this method requires too fine-grained partitioning, the discrete-event simulation will lose its advantage over discrete-time simulation. The rest of this section describes the partitioning algorithm, which uses *linear envelopes* of the function.

3.2. Linear Envelopes

To perform partitioning efficiently, we calculate *linear envelopes* of functions to estimate the range of function values. In short, a linear envelope provides a convenient way to cover possible values of a function with a linear region. Since a linear envelope of a sum of functions can be easily composed from linear envelopes of addend functions, we can cover a complex summed function with a linear envelope.

Linear envelope $\mathcal{L}(f, t_0)$ of function $f(t)$ is a region, whose edge is a set of linear equations and contains any point $(t, f(t))$ such that t is greater than a given starting point t_0 . Figure 5 shows examples of linear envelopes. In Fig. 5(a), an exponential decay function is enclosed by a linear envelope consisting of three linear inequality expressions (shown as dotted lines). In Fig. 5(b), a sine wave function is enclosed by a linear envelope consisting of four inequality expressions. Note that a linear envelope is not unique, even if $f(t)$ and t_0 are given.

It is notable that we can easily compose a linear

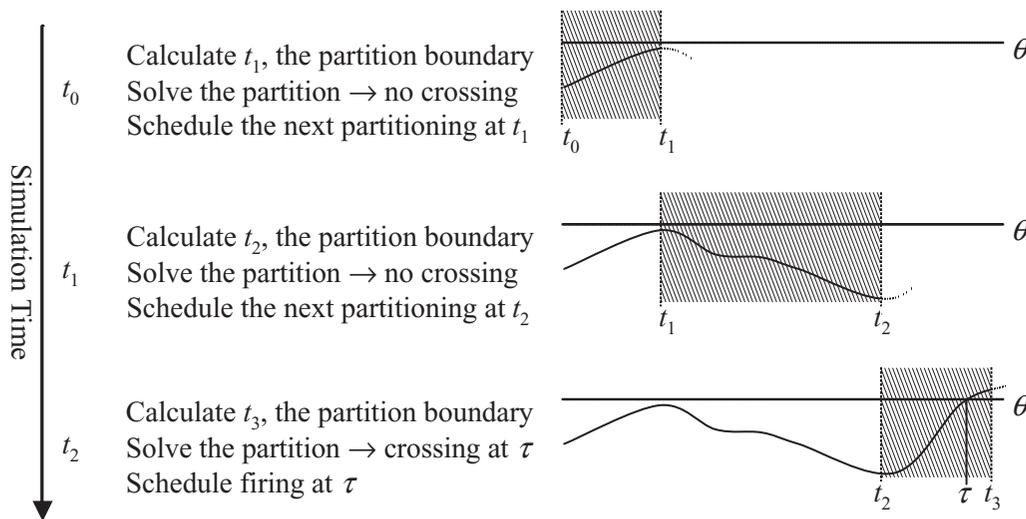


Fig. 4. Incremental partitioning method.

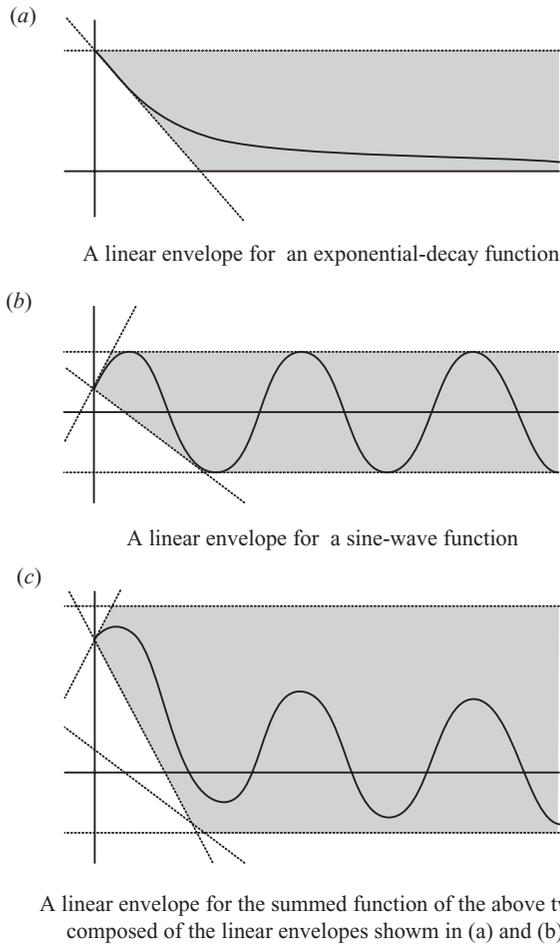


Fig. 5. Linear envelopes for nonlinear functions. Although the composed envelope shown in (c) is looser than the envelopes shown in (a) and (b), it correctly encloses the function.

envelope for a summed function of several nonlinear functions from the linear envelopes of the addend functions. Figure 5(c) shows a composed linear envelope of a function, which is a summation of the above two functions. This property enables us to calculate linear envelopes for many complex functions.

In the simulator PUNNETS, the linear envelopes are calculated using tangent gradients and their approximations. See Appendix A for the actual formulas used in the PUNNETS system.

In the following, we also use linear envelopes of the derivatives of a function. We call a linear envelope of a first-order derivative a *first-order linear envelope*, and that of a second-order derivative a *second-order linear envelope*. In need of distinction, we call a linear envelope of a non-derived function a *zeroth-order linear envelope*.

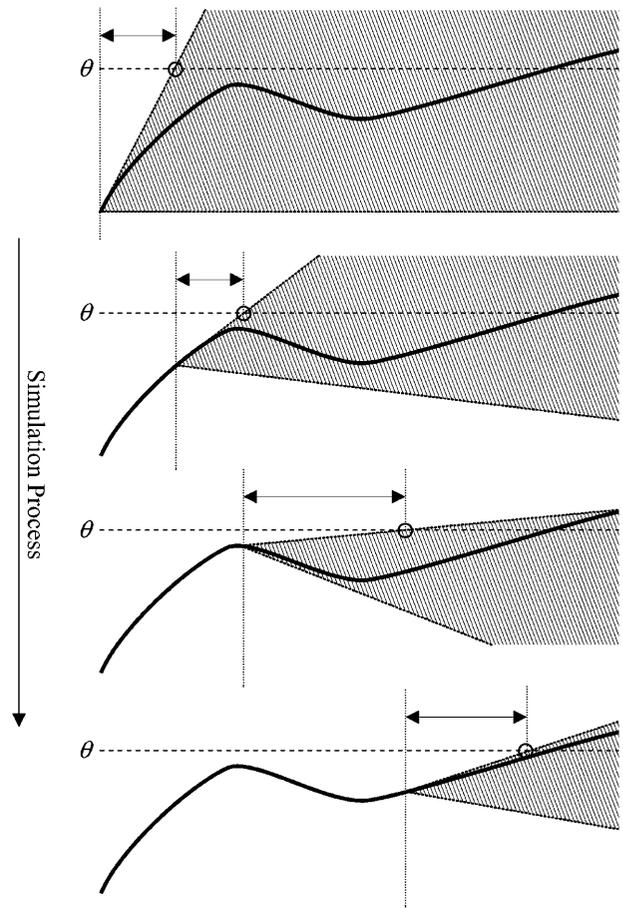


Fig. 6. Zeroth-order incremental partitioning. The arrows denote ranges of the partitions. The end of a partition is given by an intersection point of the envelope edge and the threshold (indicated by a small circle), which is in turn the start of the next partition.

3.3. Incremental Partitioning with Linear Envelopes

It is certain that a function never crosses a threshold when the threshold is out of a linear envelope of the function. We can thus partition the function at the first point where the linear envelope touches the threshold. As illustrated in Fig. 6, repeated application of this process constitutes incremental partitioning, which we call *zeroth-order incremental partitioning*.

Note that this partitioning never produces a partition that contains threshold-crossing. The closer the threshold-crossing is, the smaller the partition becomes; we never reach the threshold-crossing, as when Achilles could not catch up the turtle. One solution to escape from this paradox is to introduce a minimum partition size Δt ; in other words, fallback to discrete-time simulation. Such fallback often degrades the simulation efficiency.

A more sophisticated partitioning method uses

linear envelopes of the derived (differentiated) function. The derived function never reaches zero in a range that the linear envelope of the derivative never touches zero; in other words, the function either monotonously increases or monotonously decreases in the range. Thus, if we partition the function in the range, the partition will have, at most, one threshold-crossing point. Moreover, we can see the existence of the threshold-crossing by checking the signs of function values at both ends of the partition; if the signs are opposite, a threshold-crossing is in the partition, and at the same time, the crossing is *enclosed* in the partition so that the bisection-combined Newton-Raphson method is applicable. As shown in Fig. 7, the *first-order incremental partitioning* uses two linear envelopes, a linear envelope of the state function and a linear envelope of the derivative, and uses a larger partition from two envelopes; the method reverts to the minimum partition size Δt as before, but it relies less on the Δt fallback.

We can enlarge this approach to second-order linear envelopes as *second-order incremental partitioning*, which uses the largest partition obtained from the three linear envelopes. In a partition where the second-order derivative never touches zero, the function is either upward convex or downward convex, as shown in Fig. 7. If the function values of the both partition ends have opposite signs, we can

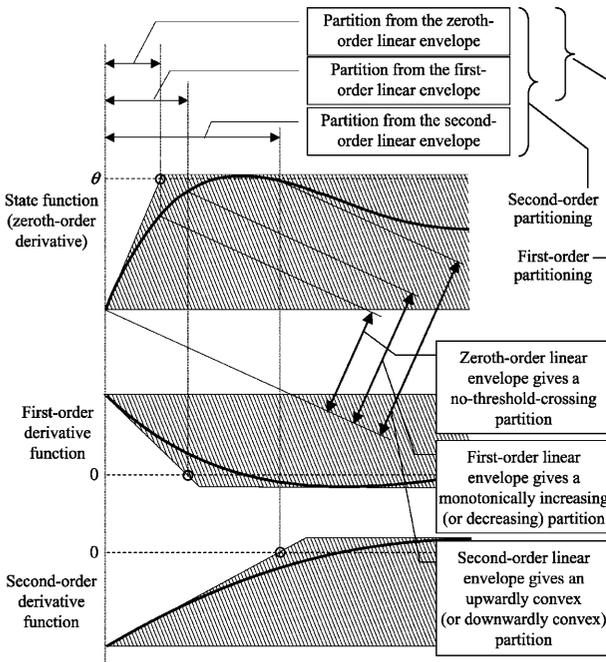


Fig. 7. First-order and second-order incremental partitioning methods. First-order partitioning uses larger one of the above two partitions, while second-order partitioning uses the largest of the three partitions.

apply the enclosed Newton-Raphson method safely. However, the problem occurs in the case with the same signs, as shown in Fig. 8(a), since the partition may have either zero or two threshold-crossings. In this case, we first discriminate the existence of the threshold-crossings by enclosed peak searching with parabola approximation [10] (see Fig. 8(b)). If the peak is beyond the threshold, we can enclose the crossing between an end of the partition and the peak; otherwise, it is analytically discriminated that the partition has no crossings. Because of the convexness of the function, the discrimination can be finished before the real peak is found (see Fig. 8(c)).

It is noteworthy that the effects of the three envelopes are complementary. When the function value is far from the threshold, the zeroth-order linear envelope usually makes the best and the largest partitioning. In the case that the function value is close to the threshold and the gradient is large, a first-order linear envelope gives the large partition that encloses the crossing. If the function value is close to the threshold and the gradient is also small, the second-order linear envelope will

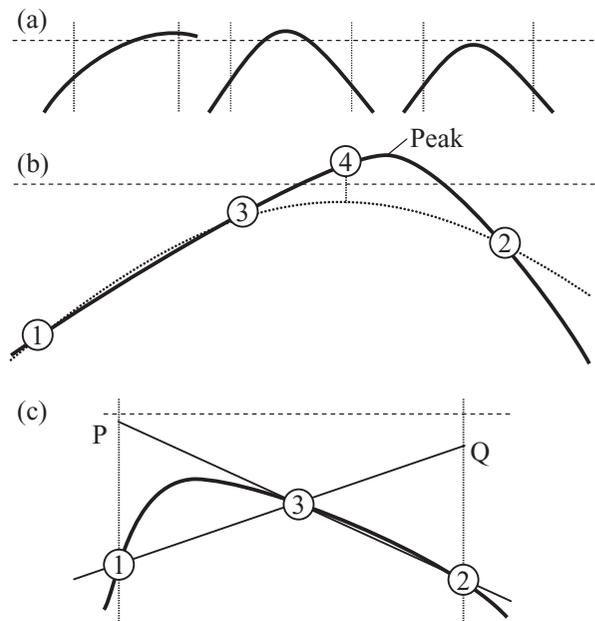


Fig. 8. Discrimination of threshold crossing. (a) Existence of threshold-crossing. Signs of the function at both ends of the partition show the left case has crossing, but cannot discriminate the middle and the right cases. (b) Enclosed peak searching. The triplet of points 1, 2, and 3 are said to be enclosing the peak ($f(x_1) < f(x_3)$, $f(x_3) > f(x_2)$, $x_1 < x_3 < x_2$). Parabola approximation (dotted line) suggests the peak point as 4, so we can narrow the enclosing to points 3, 2, and 4. The peak can be found by repeating this process. Actually point 4 is above the threshold, thus no more peak searching is required (crossing is enclosed between 3 and 4). (c) Discrimination of threshold-crossing using convexness. Since both points P and Q are below the threshold, this function has no threshold-crossing in the enclosed region. In this case, we can safely abort the enclosed peak searching.

make a partition that contains the convex curve to be solved by peak search. Since it is theoretically possible that all three approaches may fail to produce a good partition, it is still necessary to revert to the minimum partition size Δt ; however, this rarely happens in actual simulation.

3.4. Applicability of the Incremental Partitioning Method

The incremental partitioning method uses linear envelopes. For calculation of a second-order linear envelope, the function must be second-order differentiable. Note that any second-order differentiable function satisfies C_1 -class continuity, that is, the requirement of the Newton-Raphson method.

Moreover, to perform partitioning effectively, the vertical range of linear envelopes at a given starting point t is expected to converge to point $(t, f(t))$. This ensures that the linear envelopes give better prediction for the nearer future.

These requirements can be relaxed by introducing additional partitions. For example, if a function with discontinuities can be split into finite ranges of continuous functions by additional partitions, the function can be handled by the incremental partitioning method. Some functions such as $f(t) = t^2$, which is unable to maintain convergence of the linear envelopes to the starting point, can be split by additional partitions to satisfy the convergence expectation.

As a result, the incremental partitioning method can be applied to any function splittable into finite ranges of second-order differentiable functions. Although the method is unapplicable to some ill-natured functions (such as a function with an infinite number of discontinuities in a finite range), we can say the method covers any arbitrary function for the purpose of neural network simulation.

4. Efficient Simulation Techniques

The previous section introduced the incremental partitioning method, which predicts the delayed firing for a neuron model with practically any arbitrary function. However, naive application of the method causes inefficient simulation. Since the prediction is based on an assumption that no further pulses arrive, it has to be updated each time a new pulse arrives at the neuron. This degrades the performance of the simulation. Moreover, the update of the prediction changes the time of the scheduled events, which stresses the scheduling mechanism.

We have developed an efficient technique that solves these problems: *quick filtering by maximum gradient checking*. It utilises the next-known-pulse arrival to suppress redundant predictions. It also suppresses the changes to scheduled events, so it reduces the simulation cost.

4.1. Quick Filtering by Maximum Gradient Checking

The incremental partitioning algorithm predicts the delayed firing time of a neuron in the case that the neuron receives no more pulses after the last delivered event. However, it is often the case that the next pulse arrival is already scheduled but not delivered yet. Utilising this information, we can decrease the cost of the prediction.

Quick filtering is a technique that uses the time of the next-known-pulse arrival to filter out unnecessary predictions. The prediction based on linear envelopes can be suppressed if we can confirm that no threshold-crossing occurs till the arrival of the next known pulse. In such a case, it is not necessary to schedule the end of the partition as an event, since the state of the neuron is anyway recalculated at the time of the next-known-pulse arrival. If the confirmation is efficient enough, the decrease of the cost of prediction and scheduling exceeds the additional cost of confirmation.

For this purpose, we introduce *zeroth-order linear envelope checking* and *gradient limit checking*. The work of zeroth-order linear envelope checking is to check the precedence of the next-known-pulse arrival by using a zeroth-order linear envelope; in this case, the calculation cost of first- and second-order linear envelopes can be suppressed. However, to reduce the cost of calculating zeroth-order linear envelopes, we introduce a quicker checking method that uses the upper limit of the gradient of the function for quick checking. Since the upper limit of the gradient is a constant for each function, it can be calculated before starting simulation. Moreover, the upper limit of the summed function can be easily calculated by summing up the upper limits of the gradients of component functions.

Figure 9 shows an example of our quick filtering method. When the next-known-pulse arrival is close to the current time (which is often the case in handling pulse bursts), the pulse arrives before the gradient upper limit line reaches the threshold. The filtering technique thus reduces the cost of re-scheduling as well as the cost of calculating linear envelopes.

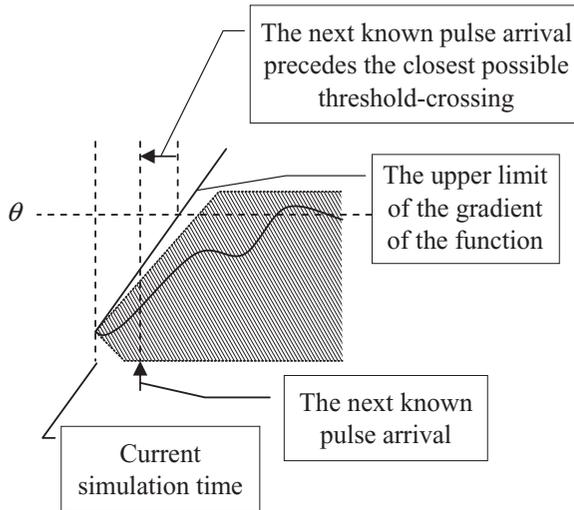


Fig. 9. Filtering a redundant prediction by gradient-limit checking.

4.2. Queuing Model for Quick Filtering

Many discrete-event simulators have a single queue to schedule all events, e.g. pulse arrivals. However, in a single-queue model, it is difficult to find the next pulse arrival for a specified neuron. To apply the quick-filtering technique, another queuing model should be used to allow a quick retrieval of the information of the next-known-pulse arrival.

To meet this requirement, we introduce another queuing model, in which each neuron has a local event queue. The local queue of a neuron holds all pending events that affect the neuron. A main schedule queue keeps neurons sorted according to the first event time of their event queues. In this model, the next event of a neuron can be easily found at the top of the neuron's local event queue.

Note that this change of queuing model does not increase the order of scheduling cost. The complete binary tree (heap tree) algorithm, which is a most popular and empirically efficient algorithm for a priority queue [11], needs a cost of $O(\log n)$ for insertion and retrieval of an entry, where n is the entries in the queue. Suppose a neural network has N neurons and each neuron has v pending events. In the single-queue model, the insertion/retrieval cost is $O(\log vN)$. On the other hand, in the object-queue model, we generally need to insert both the neuron's queue and the main queue, which keep v and N entries, respectively. The total insertion/retrieval cost is $O(\log v + \log N)$, whose order is equivalent to $O(\log vN)$, the cost of the single-queue model.

5. Implementation

We implemented PUNNETS, the pulsed neural network simulator, using the techniques described in this paper. The simulator is a 3000-step C++ program library, which is highly object-oriented and easily used by C++ programs. PUNNETS has a class that simulates any neuron based on Spike-Response model, as well as an optimised version of classes simulating an integrate-and-fire neuron with a dynamic threshold. Since neurons and synapses are designed as an object, a user can use various styles of neurons and synapses, including stochastic neurons and dynamically learning synapses. The library also has a logging ability to record the behaviour of neurons as either event reports or state graphs.

6. Experiments

We performed a series of experiments to prove the efficiency of the incremental partitioning method and the quick filtering technique. In the experiments, we used 10^{-8} as the value of ϵ (the minimum movement of x for one iteration of Newton-Raphson method).

Figure 10(a) shows the zeroth-order incremental partitioning on a summed function consisting of a sine function and an exponential function. In this figure, the simulator makes 19 partitions, although the later partitions are too narrow to see. The last eight partitions are enlarged in Fig. 10(b). Before the Δt -cutoff is used, the distance between the function and the threshold reaches less than the epsilon value and causes firing. If we use first- or second-order incremental partitioning, the area shown in Fig. 10(b) is partitioned into only one partition. In this case, nine iterations of the Newton-Raphson method correctly find the firing time. The gaps between firing times of zeroth-, first-, and second-order partitioning are less than 10^{-8} . It is clear that the precision achieved by discrete-event simulation outperforms discrete-time simulation, which requires 10^{11} synchronous updates to achieve 10^{-8} precision in a 10^3 temporal range.

Note that the number of iterations in the Newton-Raphson method (nine times) was almost the same as the number of partitions in zeroth-order partitioning (eight times). This is because, in this local range, the gradient of the edge of the linear envelopes is near to the tangent of the function, so that the zeroth-order partitioning makes the same movement step as the Newton-Raphson method does. However, higher-order partitioning has the advantages that, first zeroth-order partitioning lacks the general solv-

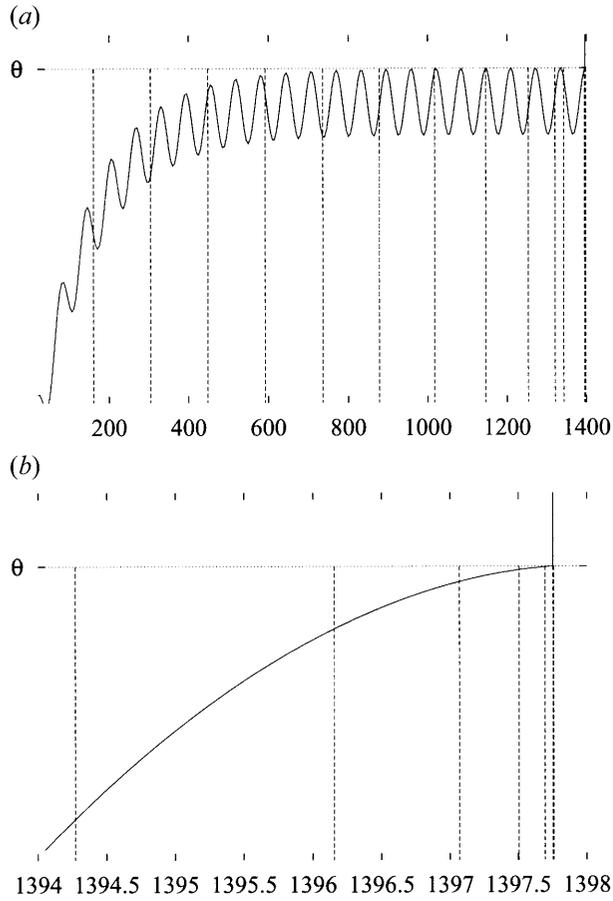


Fig. 10. Application of zeroth-order incremental partitioning. (a) Incremental partitioning for the sum consisting of an exponential function and a sine function; (b) Enlarged feature around the threshold-crossing point of (a).

ing power of an equation; second, the cost of an iteration of Newton-Raphson method is lower than the cost of a partitioning.

Figure 11 shows the simulation result from a neuron model, that is, an addition of two sine waves with slightly different wavelength. This condition corresponds to one of the worst cases, since the composed linear envelope of the function becomes much broader than the actual range of the function.

Figure 11(a) shows the result from zeroth-order incremental partitioning. Despite the broad linear envelope, the simulator reaches the firing time after 152 partitions. Higher-order partitioning achieves better results: first-order partitioning shown in Fig. 11(b) requires 97 partitions, and second-order partitioning shown in Fig. 11(c) requires only 76 partitions to reach the firing time. These experiments show that the works of higher-order partitioning are complementary to the works of the zeroth-order partitioning.

We tested the performance of our method by simulating a large-scale network. The network con-

sists of 100 neurons in the Spike-Response model. Every neuron has a response function $\eta(t) = \exp(-\theta_i t)$ and sine-wave external input $H(t) = w \sin(\omega t)$, and 10 connections from other neurons, each of which has the activation function $\varepsilon_{ij}(t) = w_{ij} (\exp(-\theta_i t) - \exp(-\theta_{ij} t))$, where w_{ij} and θ_{ij} are randomly determined for each connection. We also introduced 200 random pulses to the network, so 6276 fires were observed in the range of simulation.

Table 1 lists the performance results carried out on a Pentium 4 Xeon 2-GHz processor. The second-order partitioning method with quick filtering is the fastest of all the tested configurations. The table shows that the reduction of the number of partitions by higher-order partitioning algorithms exceeds the additional cost of the complex partitioning algorithm. In addition, the quick-filtering techniques – gradient-limit checking and zeroth-order linear checking – are effective to filter out calculation of partition ends to be re-scheduled. Note that in all tests, memory consumption was kept under 1.6 megabytes.

7. Related Work

Only a few studies pursue discrete-event simulation of pulsed neural networks. The first simulator by Watts, SPIKE [6] targeted a simple neuron model and a small network. He showed the advantage of the discrete-event simulation framework by simulating complex behaviour on a hand-made neural network.

Mattia and Giudice [8] developed techniques for large-scale discrete-event simulation of pulsed neural networks. They achieve efficiency by grouping simultaneous pulse arrivals into the one event, using a layered queue structure. Their simulator efficiently handles synaptic plasticity and Poisson-distributed random inputs. They also discuss the handling of delayed firing in the case of a neuron model with a simple differential equation, but it is not applicable to a general neuron model.

Graßmann proposed a distributed simulation of pulsed neural networks on a discrete-event simulation framework [12,7]. He reported speedup by a factor of 2.4 on three CPUs. He also mentioned that delayed firing can be predicted by using table lookup, but details are not given¹.

¹ Although a table-lookup method can accelerate calculation of delayed firing if the pulses are represented by a simple formula, it seems inapplicable to a neuron with input of various pulse models and to a neuron with external inputs. Moreover, to achieve high precision for the calculation of time and threshold, our techniques will be also required, such as firing possibility check by peak search and refinement of the calculation by the Newton-Raphson method.

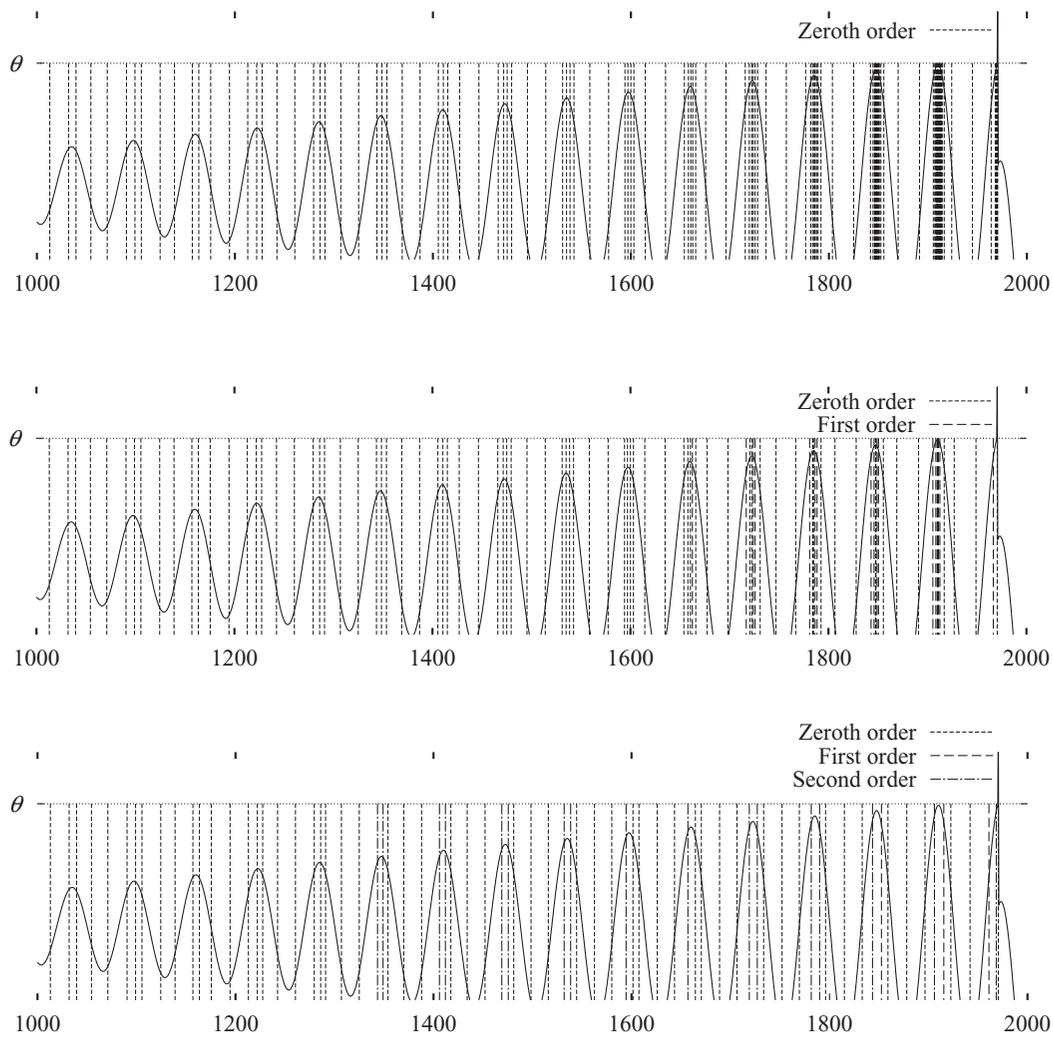


Fig. 11. Order difference of incremental partitionings.

Table 1. Performance experiments

Partitioning	2nd		1st		0th	
	Yes	No	Yes	No	Yes	No
Time (sec)	5.09	6.02	5.21	6.00	10.88	11.21
# of Partitions	53,790	90,348	55,326	91,995	142,057	179,624
0th partitions	41,222	75,904	42,292	77,051	133,811	171,448
1st partitions	11,699	13,484	13,034	14,944	0	0
2nd partitions	869	960	0	0	0	0
Δt partitions	0	0	0	0	8176	8176
# of Re-scheduled events	23,545	56,350	23,424	56,350	22,527	56,350
# of events filtered by GLC ^a	26,720	0	26,752	0	27,218	0
# of events filtered by 0th-LE ^b	9,838	0	9,917	0	10,349	0

^a Gradient-limit checking

^b Zeroth-order linear envelope checking

8. Summary and Future Work

We developed the second-order incremental partitioning method, an efficient method for predicting the delayed firing time from the function of the neuron state variable. We also devised the quick-filtering technique, which uses the time of a future pulse arrival to reduce the cost of future prediction. Using these techniques, we implemented a neural network simulator that is based on a discrete-event simulation framework but is still capable of simulating practically any Spike-Response neuron model.

One of our future works is the parallelisation of the simulator. Many large-scale discrete-event simulations are now performed in a parallel computing environment [13,14]. The discrete-event simulation of a pulsed neural network seems a suitable application for parallelisation, since every pulse transmission can be treated as an event, and delays between neurons enable us to use a simpler synchronisation mechanism. Moreover, the queuing model used in our simulation localises the scheduling information, so it is expected that the simulation can be speeded up more by parallelisation of our queuing model than by that of a single-queue model.

Appendix A: Linear Envelope Calculation in PUNNETS²

This section illustrates calculations of linear envelopes in the PUNNETS system.

A.1. Monotonic Convex Function

It is simple to calculate linear envelopes for monotonically increasing or monotonically decreasing and converging convex functions, such as $f(x) = \exp(-x)$.

For a monotonically decreasing function, the following definition of linear envelope for the point x_0 is used:

$$\mathcal{L}(f, x_0) = \begin{cases} y \leq f(x_0) \\ y \geq f(x_0) + \frac{df}{dx} \Big|_{x=x_0} (x - x_0) \\ y \geq f(\infty) \end{cases} \quad (1)$$

A monotonically increasing function can be transformed into a monotonically decreasing function,

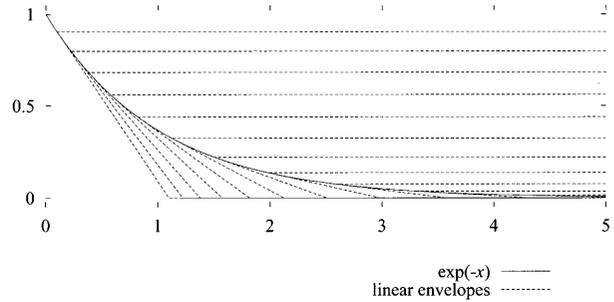


Fig. 12. Linear envelopes of $f(x) = \exp(-x)$.

such as $g(x) = -f(x)$. Figure 12 illustrates linear envelopes for various points of $f(x) = \exp(-x)$.

A.2. Sine Function

In calculation of the linear envelopes of a sine function, it is useful to use information of tangent lines from the start point of the linear envelope. However, there is no easy formula for calculating the tangent lines. We used the following approximation for the gradient of one of the tangent lines:

$$\gamma(x_0) = \begin{cases} \cos(\theta) & (\theta < -\frac{\pi}{2}) \\ \sin(\alpha(\cos(\beta(\theta + \frac{\pi}{2}))) - 1) & (\theta \geq -\frac{\pi}{2}) \end{cases} \quad (2)$$

$(\theta = x_0 + 2n\pi, -\pi \leq \theta \leq \pi)$

where $\alpha = 1.311$ and $\beta = 0.375867$. The gradient of the other tangent line can be calculated from $-\gamma(t_0 + \pi)$.

As shown in Fig. 14, the approximation always gives a greater gradient than the actual gradient of a tangent line. A linear envelope calculated from the approximation therefore always contains a linear envelope calculated from the actual tangent lines. Figure 13 illustrates linear envelopes for $f(x) = \sin(x)$. In a formula, the linear envelope is given below:

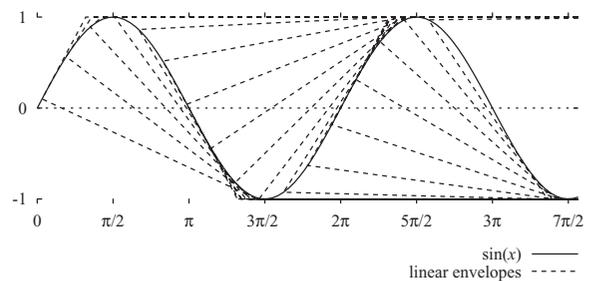


Fig. 13. Linear envelopes of $f(x) = \sin(x)$.

² Makino T. (2003) Punnets reference manual. <http://snowelm.com/~t/research/software/>

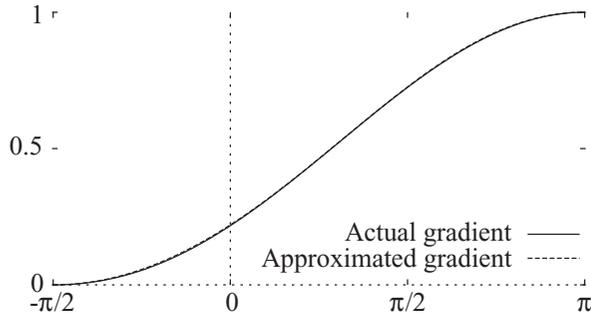


Fig. 14. Comparison of actual gradient and approximated gradients $\gamma(x_0)$.

$$\mathcal{L}(\sin, x_0) = \begin{cases} y \leq 1 \\ y \leq -\gamma(x_0 + \pi)(x - x_0) + x_0 \\ y \geq \gamma(x_0)(x - x_0) + x_0 \\ y \geq -1 \end{cases} \quad (3)$$

A.3. Pulse Response Function

In the Spike-Response model, the following function is often used as a response to a pulse:

$$s(x) = w \cdot (\exp(-ax) - \exp(-bx)) \quad (4)$$

$(0 < a < b)$

We can apply linear transformation for the function in the following canonical form:

$$S(x) = (\exp(-zx) - \exp(-z\phi x)) \quad (5)$$

$(\phi = \frac{b}{a} > 1, z = \frac{2 \log \phi}{\phi - 1})$

Here, z is a normalisation factor that fixes the inflection point of the function to $x = 1$. Beyond the point ($x > 1$), the function is monotonically decreasing, so the linear envelope described in Appendix A. 1 is applicable. This function reaches the peak at $x = \frac{1}{2}$, and the value at the peak is $S(\frac{1}{2}) = \exp(-\frac{1}{2}zx) - \exp(-\frac{1}{2}z\phi x)$.

We also use the approximation to estimate the gradient of the tangent line:

$$\xi(x_0) = \frac{-z \exp(-z) + z\phi \exp(-z\phi)}{(1 - (1 - x_0)^\psi)} \quad (6)$$

where $\psi = 0.3(\log_{10}\phi)^2 + 2.45$. Although this approximation is not as good as the approximation of sine function, the function always gives a larger gradient than that of the actual tangent.

This approximation gives the linear envelope as in follows:

$$\mathcal{L}(S, x_0) = \begin{cases} y \leq S(\frac{1}{2}) \\ y \leq S(x_0) + \begin{cases} \frac{dS}{dx}|_{x=x_0} (x - x_0) & (x_0 < \frac{1}{2}) \\ 0 & (x_0 \geq \frac{1}{2}) \end{cases} \\ y \geq S(x_0) + \begin{cases} \xi(x_0)(x - x_0) & (x_0 < 1) \\ \frac{dS}{dx}|_{x=x_0} (x - x_0) & (x_0 \geq 1) \end{cases} \\ y \geq 0 \end{cases} \quad (7)$$

Figures 15 and 16 illustrate linear envelopes of Eq. (7) for $\phi = 10$ and $\phi = 2$. Figures 17, 18, 19 and 20 show the actual tangent, estimated tangent, estimation error, and actual and estimated tangents at $\phi = 10$.

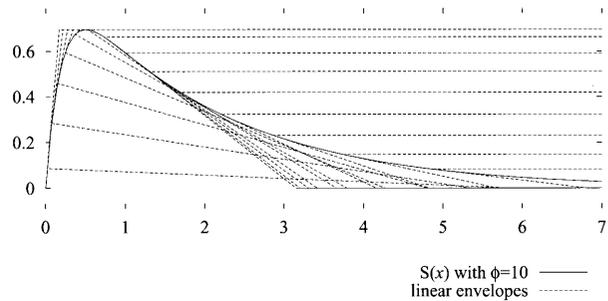


Fig. 15. Linear envelopes of $S(x)$ at $\phi = 10$.

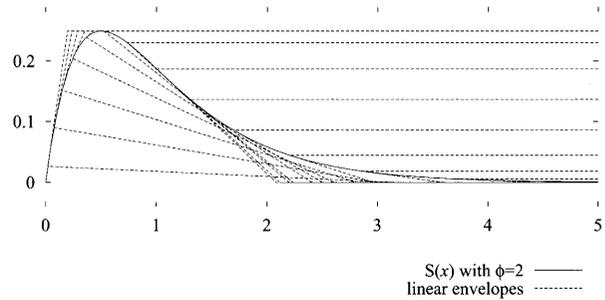


Fig. 16. Linear envelopes of $S(x)$ at $\phi = 2$.

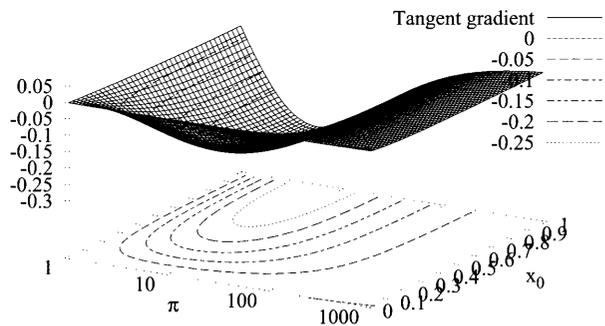


Fig. 17. Actual tangent gradient.

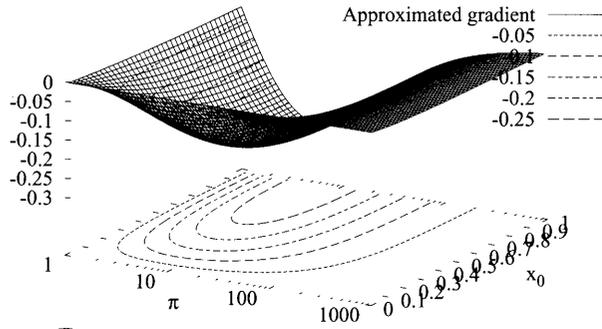


Fig. 18. Approximated tangent gradient $\xi(x_0)$.

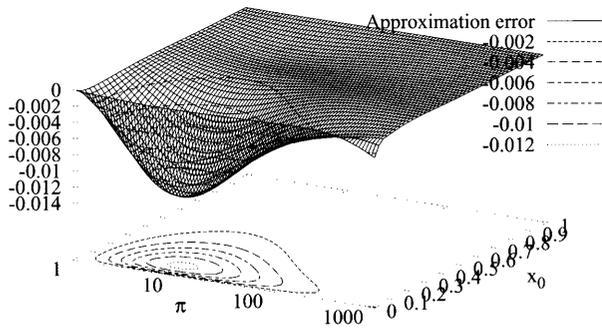


Fig. 19. Approximation error of $\xi(x_0)$.

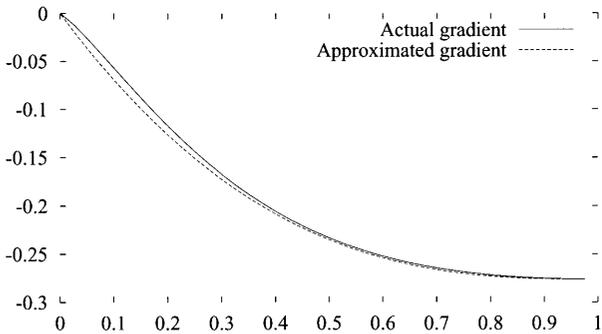


Fig. 20. Actual and approximated tangents of $\xi(x_0)$ when $\phi = 10$.

Acknowledgements. I would like to express my deep gratitude for Prof. Kazuyuki Aihara, who gave me invaluable help for this research. I thank Prof. Jun-ichi Tsujii, Dr. Takashi Ninomiya, and Mr. Yusuke Miyao for precious advices. This research is supported by Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists.

References

1. Makino T, Aihara K, Tsujii J (2001) Towards sentence understanding: Phase arbitration in temporal-coding memory mechanism. Second Workshop on Natural Language Processing and Neural Networks (NLPNN'2001), Tokyo, Japan 46–52
2. Makino T (2001) A Pulsed Neural Network for Language Understanding: Discrete-Event Simulation of a Short-Term Memory Mechanism and Sentence Understanding. PhD dissertation, Department of Information Science, Tokyo University, Tokyo, Japan
3. Fujii H, Ito H, Aihara K, Ichinose N, Tsukada M (1996) Dynamical cell assembly hypothesis – theoretical possibility of spatio-temporal coding in the cortex. *Cognitive Science* 9: 1303–1350
4. Bower JM, Beeman D (1995) The book of GENESIS: Exploring Realistic Neural Models with the GENERAL NEural SIMulation System. TELOS/Springer-Verlag, New York
5. O'Reilly RC, Munakata Y (2000) Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain. MIT Press, Cambridge, MA
6. Watts L (1994) Event-driven simulation of networks of spiking neurons. In: Cowan JD, Tesauro G, Alspector J (editors) *Advances in Neural Information Processing Systems*, 6, Morgan Kaufmann 927–934
7. Grassmann C, Anlauf JK (1999) Fast digital simulation of spiking neural networks and neuromorphic integration with SPIKELAB. *Int J Neural Systems* 9(5): 473–478
8. Mattia M, Del Giudice P (2000) Efficient event-driven simulation of large networks of spiking neurons and dynamical synapses. *Neural Computation* 12: 2305–2329
9. Gerstner W (1998) Spiking neurons. In: Maass W, Bishop CM (editors) *Pulsed Neural Networks*, MIT Press, Cambridge, MA 3–53
10. Press WH, Flannery BP, Teukolsky SA, Vetterling WT (1988) *Numerical Recipes in C*. Cambridge University Press
11. Jones DW (1986) An empirical comparison of priority queue and event set implementations. *Comm ACM* 29: 300–311
12. Graßmann C, Anlauf JK (1998) Distributed, event driven simulation of spiking neural networks. *Proceedings International ICSC/IFAC Symposium on Neural Computation (NC'98)*, ICSC Academic Press, 100–105
13. Fujimoto RM (1993) Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing* 5(3): 213–230
14. Lin Y-B, Fishwick PA (1996) Asynchronous parallel discrete event simulation. *IEEE Trans Systems, Man and Cybernetics* 26(4)