Quantized State Simulation of Spiking Neural Networks

Guillermo L. Grinblat, Hernán Ahumada and Ernesto Kofman
CIFASIS – CONICET
FCEIA. Universidad Nacional de Rosario
Argentina

Abstract

In this work, we explore the usage of Quantized State System (QSS) methods in the simulation of networks of spiking neurons. We compare the simulation results obtained by these discrete–event algorithms with the results of the discrete time methods in use by the neuroscience community. We found that the computational costs of the QSS–methods grows almost linearly with the size of the network, while it grows at least quadratically in the discrete time algorithms. We show that this advantage is mainly due to the fact that QSS methods only perform calculations in the components of the system that experience activity.

1 Introduction

Continuous System Simulation is a topic that has rapidly evolved in the last decades. The development of faster and more powerful computers has allowed the representation of larger and more complex models which require the usage of more efficient and sophisticated simulation algorithms.

Hundreds of numerical integration methods for ordinary differential equations (ODEs) can be found in the literature [10, 11, 30, 7]. These algorithms, according to their features, are classified as one–step or multistep; implicit or explicit; fixed or variable step; fixed or variable order; etc. In spite of their differences, all the methods share a property: they are all based on time discretization, i.e., given the solution at a time instant, they compute the approximate solution for the next discrete time point.

In the last years, a new family of algorithms was developed that replace the time discretization by the quantization of the state variables. Based on an original idea of Bernard Zeigler, who showed that Continuous Time Systems with its inputs and outputs being quantized can be represented by DEVS models [38, 40, 39], a general method for numerical integration of ODEs called Quantized State System (QSS) was proposed in [22].

The formulation of the QSS–method was followed by second and third order accurate algorithms (QSS2 [18] and QSS3 [20], respectively). The QSS family has nice stability, convergence and error bound theoretical properties [22, 18, 7, 29], and, from a practical point of view, offers important advantages to detect and handle discontinuities [19].

QSS–methods are also very efficient to simulate large sparse systems [18, 7], since they only invoke calculations on the states that experience sensible changes in their values or derivatives. In other words, QSS algorithms intrinsically exploit the *activity* of the system [17, 26, 28].

In this work, we explore the usage of QSS methods to simulate deterministic ODE models of spiking neural networks (SNN). These models are emerging as a plausible paradigm for characterizing neural dynamics in the cerebral cortex. The SNN models have high biological fidelity, and can model many characteristics of brain architecture [5].

SNN models are usually large (they are composed by several neurons), sparse (each neuron is generally connected to a small subset of neurons), and each spike represents a discontinuity in the ODE. The presence of discontinuities and its large dimension poses several difficulties to the different conventional numerical methods used by SNN community. Consequently, the simulation of large networks of spiking neurons becomes slow and demands for more suitable numerical methods.

Taking into account that QSS methods are good at discontinuity handle and sparsity exploitation, they are –in principle– good candidates to improve the performance of SNN simulation. This hypothesis is reinforced by the fact that previous works have shown some advantages of using the DEVS methodology in the simulation of non–ODE models of SNN [32, 4]. Also, in [36] a discrete version of Hodgkin-Huxley model (which could be used to model SNN) is efficiently implemented with Cell–DEVS and a similar approach, which approximates the continuous time behavior, is reported in [37].

The goal of this work is then to implement simulations of ODE models of SNN based on QSS methods and to compare their performance with those of conventional numerical methods currently in use for those models.

As a main contribution, we shall show that the principal advantage of using QSS methods to simulate SNN is that the computational costs grow linearly with the number of neurons in the network, while classic integration techniques lead to quadratic growth (in the best case). In large networks, this property is translated into a sensible reduction of the CPU time required to complete a simulation.

The article is organized as follows: Section 2 provides the background for the work, presenting different ODE models for SNN, analyzing the different numerical methods used in the literature, and introducing the QSS methods as well as the software tools for their implementation. Section 3 then describes the work done to model and to simulate SNN in a software tool that implements the complete family of QSS algorithms. After that, Section 4 presents the simulation results and compares those obtained with different QSS methods with those obtained by classic methods for different SNN configurations. Finally, Section

5 finishes the work with conclusions and some ideas to continue with this line of research.

2 Modeling and Simulation of Spiking Neural Networks

This section provides the background on which the rest of the work is based. We first introduce different ODE models of SNN and then we present the classic numerical methods used to simulate them. After that, we introduce the family of QSS methods and the software tools that implement them.

2.1 Models of Spiking Neural Networks

A single spiking neuron can be described by a system of ODEs with discontinuities at the firing times. Several models have been proposed, with varying complexity. Among the most used we can mention the following ones:

- Hodgkin-Huxley model [12]: This model was developed in the 1950s based on the experiments performed on the squid giant axon. Due to this fact, the model parameters have a clear observed meaning. The main inconvenience is that its simulation is very expensive due to the model complexity, as each neuron is represented by four equations governed by ten parameters. Thus, its usage is limited to networks formed by few neurons [15].

- Integrate and Fire [23, 34]. Contrary to Hodgkin-Huxley, this is an extremely simple model. A neuron is modeled by the equation

$$\dot{v}(t) = I(t) + a - bv(t), \quad \text{if } v(t) \geq v_u \text{ then } v \leftarrow c,$$

  where $v(t)$ is the membrane potential, $I(t)$ is input current, a, b, c and v_u are user defined parameters to obtain different behaviours.

  Due to its simplicity, the equation can be analytically integrated between discontinuities and large neural networks can be simulated. However, the model is not rich enough to represent many features that are usually observed in real neurons.

  Several modifications have been proposed to improve this model, such as the inclusion of a quadratic term on $v(t)$ (quadratic Integrate and Fire [9]), or the addition of a second state variable in order to represent more complex behaviours (Integrate and Fire or Burst [33]).

- Izhikevich model [14, 16]: Recently, Izhikevich proposed a rather simple but versatile model that can represent different behaviours according to their parameter configuration. Its equations are:

$$\begin{aligned} C\dot{v} &= kv(v - v_t) - u + I \\ \dot{u} &= a(bv - u) \end{aligned} \tag{1}$$

Where $v(t)$ represents, as before, the membrane potential and $u(t)$ models the membrane restitution phenomenon. Variable $I(t)$ is the input current and the rest are user defined constant parameters that allow obtaining different types of behaviour.

When at time t variable $v(t)$ reaches a given threshold value, the firing is produced and the state variables are reset as follows:

$$\begin{aligned} v(t^+) &= c \\ u(t^+) &= u(t) + d, \end{aligned} \tag{2}$$

where c and d are user defined constant parameters.

The parameters of Eq.(1) are usually selected so that the equation becomes

$$\begin{aligned} \dot{v}(t) &= 0.04 \cdot v^2 + 5 \cdot v + 140 - u + I(t) \\ \dot{u}(t) &= a \cdot (b \cdot v - u) \end{aligned} \tag{3}$$

As it can be observed in the three models described, there is always a trade–off between simulation costs and the possibility of reproducing the different behaviours observed in real neurons. There are several other models for SNN and a comparison among them can be found in [15].

We shall use Izhikevich's model along this work, as it combines rich dynamics with fair computational effort requirements.

The interconnection between neurons (i.e., synapses) can be also modeled in different ways. In this work, we selected the synaptic current approach [1, 5], also taking into account excitatory and inhibitory currents [35].

Thus, the original Izhikevich's model of Eq. (1) was transformed into Eq. (4):

$$\begin{aligned} C\dot{v} &= kv(v - v_t) - u - \eta(v - E_\eta) - \gamma(v - E_\gamma) + I \\ \dot{u} &= a(bv - u) \end{aligned} \tag{4}$$

Here, η and γ are the excitatory and inhibitory conductances, respectively; while E_η and E_γ are the *reversal potentials*. When a neuron receives the firing of an excitatory neuron, the excitatory conductance η increases its value in 6 nS, while when the firing comes from an inhibitory neuron then γ is increased in 67 nS. The rest of the time, η and γ decay exponentially following a first order dynamic:

$$\begin{aligned} \dot{\eta} &= -\lambda_\eta \cdot \eta \\ \dot{\gamma} &= -\lambda_\gamma \cdot \gamma, \end{aligned} \tag{5}$$

where λ_η and λ_γ are the decay rate parameters.

2.2 Numerical Integration of SNN Models

When a classic numerical method is used to simulate a SNN a problem appears. The models of SNN exhibit a discontinuous behaviour each time a neuron fires.

Conventional numerical methods cannot integrate across a discontinuity. The numerical integration methods are always based on polynomial approximations of some functions; which are no longer valid when those functions are discontinuous. Thus, when they perform an integration step that jumps through a discontinuity, the error committed is unacceptable. To solve this problem, the methods must detect the discontinuity first, advance the simulation time until that instant, and then restart the simulation from the new situation [7].

SNN discontinuities belong to the *state event* type, i.e., their occurrence depends on a condition on the state variables (typically, a zero–crossing of some signal). The detection of this type of discontinuities requires iterations, and it is computationally expensive. The simulation restart is also time–consuming, as the simulation step size must be restarted, typically from a small value.

From a practical point of view, the interesting SNN models are those composed by several neurons. As firings at different neurons occur at different times, the rate of firings in a network grows linearly with the number of neurons.

This is, if each neuron provokes on average 100 firings per second, a network of 1000 neurons will provoke about 100000 firings per second. Then, any classic numerical method simulating a single neuron will have an upper bound for its step size of 1/100 in order to properly handle discontinuities (accuracy and stability considerations may impose a lower upper bound). However, when simulating a network of 1000 neurons that upper bound will be as low as 1/100000. Also, each step will be 1000 times more expensive as it involves the evaluations on the derivatives of 1000 times more states.

Due to these facts, the computational cost grows at least quadratically with the number of neurons in the network.

Yet, classic numerical methods are widely used for simulating SNN. Among them, Runge–Kutta and Bulirsch-Stoer [30] appear frequently reported in the literature.

A recent work proposes a discrete event like solution based on the linearization of the ODE and its analytical solution between firings [41], yet for neuron models involving more than one state variable this approach is only first order accurate. Thus, for the case of Izhikevich's model –which is a second order system– this approach cannot offer decent results. The fact that the algorithm results only first order accurate implies that it will not be able to achieve an acceptable accuracy without increasing enormously the computational costs. However, it is worth mentioning as this solution, called *voltage stepping*, has some connection with the Quantized State System methods that we describe below.

2.3 Quantized State System Simulation

While all classic methods are of *discrete time* type, a new approach was recently developed which replaces time discretization by state quantization.

The first of these algorithms was the Quantized State System method of first order (QSS1), which is defined below.

Consider a time invariant ODE:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{6}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the state vector and $\mathbf{u}(t) \in \mathbb{R}^m$ is an input vector, which is a known piecewise constant function.

The QSS1 method [22, 7] analytically solves an approximate ODE, which is called Quantized State System:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t)) \tag{7}$$

where $\mathbf{q}(t)$ is a vector of quantized variables which are quantized versions of the state $\mathbf{x}(t)$. Each component $q_j(t)$ of $\mathbf{q}(t)$ follows a piecewise constant trajectory, related with the corresponding component $x_j(t)$ of $\mathbf{x}(t)$ by a hysteretic quantization function so that[1]

$$q_j(t) = \begin{cases} x_j(t) & \text{if } |q_j(t^-) - x_j(t)| = \Delta Q_j \\ q_j(t^-) & \text{otherwise} \end{cases} \tag{8}$$

and $q_j(t_0) = x_j(t_0)$. This is, $q_j(t)$ only changes when it differs from $x_j(t)$ by $\pm\Delta Q_j$. The magnitude ΔQ_j is called quantum. Figure 1 shows a typical QSS1 quantization function.

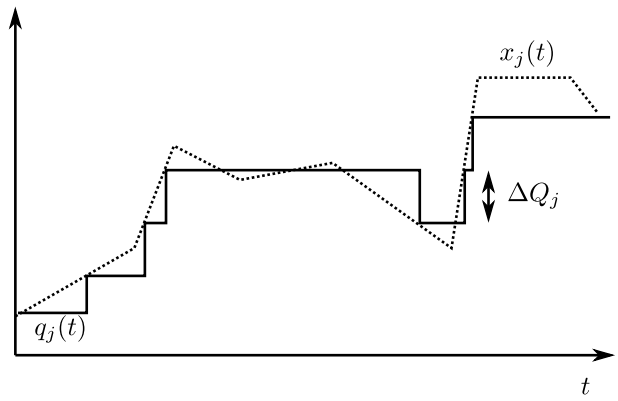

Figure 1: Hysteretic quantization

The piecewise constant evolution of the quantized variables $q_j(t)$ implies that the state derivatives $\dot{x}_j(t)$ follow piecewise constant trajectories, and then the

[1] We denote $q_j(t^-) = \lim_{\tau \to t^-} q_j(\tau)$, i.e., the limit from the left of $q_j(t)$.

states $x_j(t)$ evolve in a piecewise linear way. These facts permit the analytical integration of the system of Eq.(7) in a straightforward manner.

The QSS1 method has some nice stability and global error bound properties [18, 7, 29]. Yet, it performs only a first order approximation and a good accuracy cannot be obtained without significantly increasing the number of steps.

A second order accurate method called QSS2 was proposed in [18]. QSS2 has the same definition of QSS1, except that the components of $q_j(t)$ are now calculated to follow piecewise linear trajectories. Figure 2 shows a typical evolution of state and quantized variables.

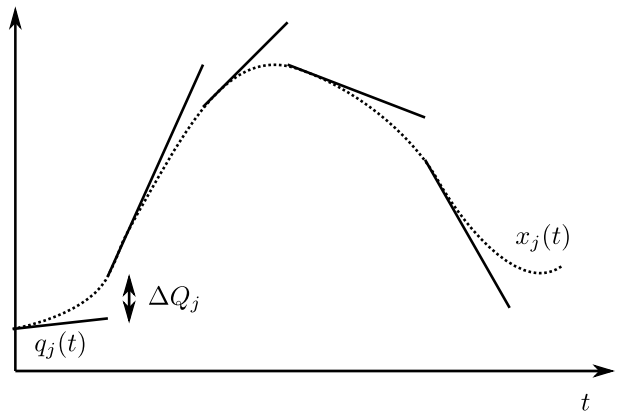

Figure 2: QSS2 quantization

In QSS2 the state derivatives $\dot{x}_j(t)$ are computed as piecewise linear trajectories and then the states $x_j(t)$ follow piecewise parabolic trajectories. Like in QSS1, the analytical solution of the quantized system of Eq.(7) can be easily obtained following a simple algorithm.

QSS1 and QSS2 ideas were also extended to obtain a third order accurate method called QSS3 [20]. In QSS3, quantized variables follow piecewise parabolic trajectories and states are piecewise cubic.

Both methods, QSS2 and QSS3, share the stability and error bound properties of QSS1. The family of QSS is completed with a Backward QSS algorithm (BQSS) and the Centered QSS (CQSS) conceived to integrate stiff and marginally stable systems, respectively [6].

Since Izikhevich's model is not stiff nor marginally stable, we shall use QSS3 as it offers the maximum accuracy order among all the QSS methods. Thus, it will obtain the best relationship between accuracy and computational costs.

The quantization functions shown in Figs.1–2 use a uniform quantum ΔQ_j. Alternatively, QSS methods can use logarithmic quantization, where the quantum is proportional to the corresponding state magnitude [21].

When uniform quantization is used, the absolute global simulation error is bounded by a linear function of the quantum [7]. In the case of logarithmic quantization, the relative global simulation error is intrinsically controlled [21].

Steps in QSS methods are only produced when some quantized variable $q_j(t)$ changes, i.e., when the corresponding state $x_j(t)$ differs from $q_j(t^-)$ in a quantum. That change implies also that some state derivatives (those that depend on x_j) are also changed. Then, each step involves a change in only one quantized variable and in some state derivatives.

Thus, when a large sparse system experiences activity only in a few states while the rest of the system remains unchanged, the QSS methods intrinsically exploit this fact performing computations only when and where the changes occur.

Another important advantage of the QSS methods is that they handle discontinuities in a straightforward and very efficient manner [19]. According to the order of the method, the state variables follow piecewise linear, parabolic or cubic trajectories. Then, detecting zero crossings is straightforward, as it involves solving a cubic equation in the worst case. Once a discontinuity is detected, the algorithm handles it as an ordinary step, since each step is in fact a discontinuity in a quantized variable. Hence, the occurrence of a discontinuity implies only some local calculations to recompute the state derivatives that are directly affected by that event.

2.4 QSS Methods and DEVS

Although QSS algorithms can be easily coded in any programming language, they are usually implemented as *discrete event systems* within the DEVS formalism framework [39].

Notice that each component of Eq.(7) can be considered as the coupling of two elementary subsystems. A static one,

$$d_j(t) = f_j(q_1, \cdots, q_n, u_1, \cdots, u_m) \tag{9}$$

and a dynamical one

$$q_j(t) = Q_j(x_j(\cdot)) = Q_j(\int d_j(\tau)d\tau) \tag{10}$$

In the case of the first order QSS1 method, Q_j is the hysteretic quantization function (it is not a function of the instantaneous value $x_j(t)$, but a functional of the trajectory $x_j(\cdot)$).

Since the components $u_j(t)$ and $q_j(t)$ are piecewise constant, the output of Subsystem (9), i.e., $d_j(t)$, will be piecewise constant. In this way, both subsystems have piecewise constant input and output trajectories.

If we represent every change of a piecewise constant trajectory as an *event*, then the trajectories can be considered sequences of events.

Thus, the sub–systems of Eqs.(9) and (10) can be seen as discrete event systems that process event sequences.

The DEVS formalism [39] allows to describe any model that processes event sequences and the representation of the models of Eqs.(9) and (10) is in fact very simple. The DEVS equivalent of Eq.(9) is called *static function* and the DEVS

equivalent of Eq.(10) is called *hysteretic quantized integrator*. Their definition can be found in [7].

Figure 3 shows the block diagram representation of Eq.(7). It is composed by n static subsystems like that of Eq.(9) and n dynamic subsystems like Eq.(10).

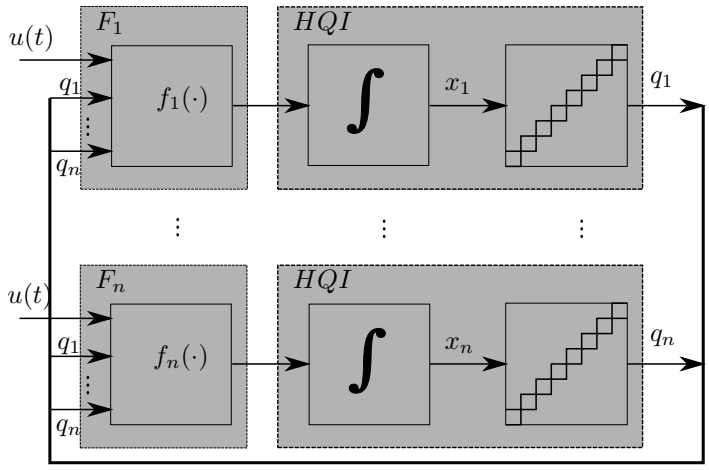

Figure 3: Block Diagram representation of a QSS1 approximation.

Connecting the DEVS models corresponding to static functions and quantized integrators following the block diagram of Fig.3, a coupled DEVS model is obtained that exactly represents the dynamics of Eq.(7).

The same idea can be applied to represent QSS2 approximations as DEVS models, but now the events carry two variables with the initial value and slope of each segment of a piecewise linear trajectory. The DEVS models corresponding to static functions and quantized integrators are more complex since they take into account also the slopes of the corresponding trajectories.

The QSS3 method can be also implemented following these ideas.

The family of QSS methods, including QSS1, QSS2, QSS3, BQSS and CQSS algorithms were implemented in PowerDEVS [3], a DEVS–based simulation software.

Figure 4 at the left shows the QSS continuous library of PowerDEVS. The blocks contained in this library implement the quantized integrators and static functions for the mentioned QSS methods.

Hybrid systems can be represented and simulated in PowerDEVS coupling blocks of the continuous library with the blocks of the QSS hybrid library shown at the right of Figure 4. The hybrid blocks are DEVS models that handle different types of discontinuities making use of the piecewise polynomial features of quantized variable trajectories in QSS methods.

For instance, the hybrid system corresponding to Izhikevich's model of Eqs.(3)–(2) can be implemented by the PowerDEVS model of Figure 5.

The models of continuous and hybrid systems in PowerDEVS can be built

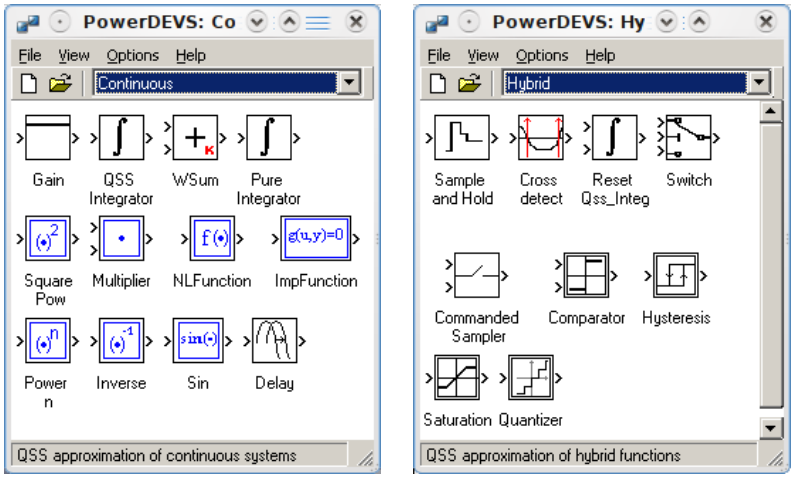

Figure 4: PowerDEVS QSS Continuous and Hybrid Libraries.

using the classic Block Diagram approach. This is, we include an integrator for each state variable and then we build the expression of the state derivatives using continuous and/or hybrid blocks. Additionally, for input signals, we use source blocks.

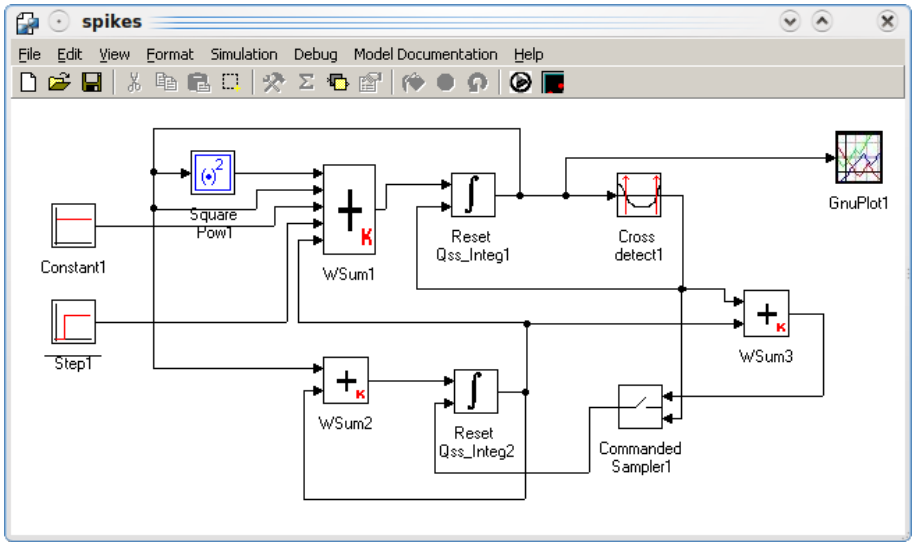

Figure 5: Izhikevich's model in PowerDEVS.

The first order accurate QSS1 algorithm was also implemented in other simulation tools: CD++ [8], VLE [31], and Dymola/Modelica [2]. However, those implementations do not include higher order methods like QSS3 (with the ex-

ception of ModelicaDEVS, but this implementation is not efficient enough [2]), so we shall focus only on PowerDEVS.

3 PowerDEVS Modeling and Simulation of SNN

This section describes the work done to simulate SNN in PowerDEVS using QSS methods. We first describe the PowerDEVS model of a single isolated neuron, then the model of a neuron with synapsis interface, and finally the model of a network of neurons and the modifications introduced in PowerDEVS in order to efficiently simulate large coupled systems.

3.1 PowerDEVS Model of a Single Isolated Neuron

A single isolated neuron, following Izhikevich's model of Eqs.(1)–(2), was modeled in PowerDEVS as Figure 5 shows. This Block Diagram was built following the classic procedure mentioned above, and it is the direct representation in Block Diagrams of the corresponding differential equations.

There, the blocks 'Reset QSS_Integ1' and 'Reset QSS_Integ2' compute the states v and u, respectively. Similarly, the static blocks 'Wsum1', 'Square Pow1', and 'Wsum2' calculate the state derivatives, making use also of the source blocks 'Constant1' and 'Step1'. The block 'Cross detect1' produces an event each time its input signal crosses a given level, in this case when the condition $v = 60$ is reached. This event is used to reset the first integrator to the value $v = -30$ and to compute the signal $u - 30$ and reset the second integrator.

3.2 PowerDEVS Model of a Neuron with Synapsis Interfaces

The addition of the synapsis ports transforms Eqs.(1)–(2) into Eqs.(4)–(5). Thus, we modified the PowerDEVS model of Fig.5 by adding new blocks as Fig.6 shows.

In this new PowerDEVS block diagram, the new blocks 'Reset QSS_Integ3' and 'Reset QSS_Integ4' calculate the new states η and γ, and the blocks 'Gain1' and 'Gain2' compute the corresponding state derivatives. The blocks 'Commanded Sampler2' and 'Commanded Sampler3' are in charge of reseting the integrators of η and γ when they receive the events that indicate the firing of other neurons.

The blocks 'WSum4', 'WSum5', 'Wsum6', 'Multiplier1', and 'Multiplier2' calculate the additional terms of the derivative $\dot{v}(t)$ at the right hand side of Eq.(4), and finally, the blocks 'WSum7' and 'WSum8' compute the reset values for η and γ.

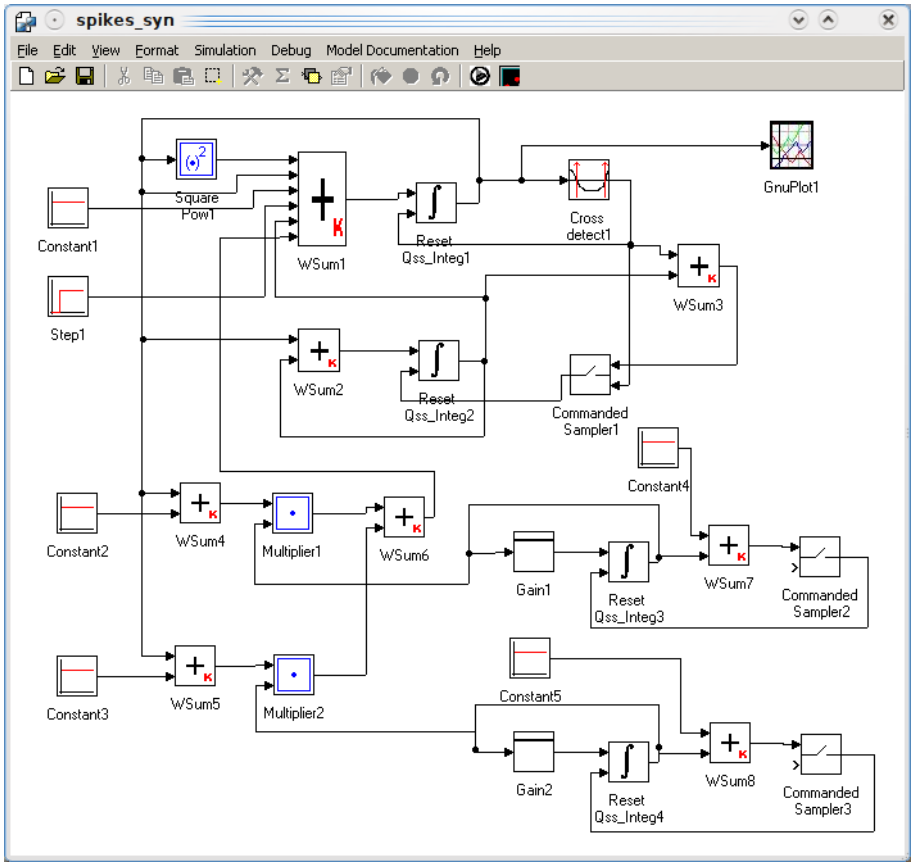

Figure 6: Izhikevich's model with synapsis in PowerDEVS.

3.3 Modeling and Simulation of a Large Neural Network

Modeling and simulating a large network of neurons like those of Fig.6 using PowerDEVS requires to solve some problems first.

3.3.1 Model Construction

The first of the problems is related to the construction of a very large model with a graphical user interface like that of PowerDEVS. Although it allows to encapsulate complex models so that the entire block diagram of Fig.6 appears as a single block, copying and coupling 1000 of these blocks (in order to simulate a network of 1000 neurons) is impossible.

We also wanted to generate the connection structure in a random way in order to perform multiple simulation runs of different networks and the graphical user interface of PowerDEVS does not have that capability.

So, we used the PowerDEVS GUI only to build the model of a single neuron

(with and without synapsis ports) and then we wrote a C++ program that replicates that model as many times as we want and that generates random interconnections (if necessary) between the different neurons.

3.3.2 Model Simulation

The second problem we faced was related to the size and the structure of the model and the way PowerDEVS searches the next event time and the next transitioning model.

Notice that the model of a single neuron with synapsis interfaces (Fig.6) contains 27 blocks (without taking into account the Gnuplot block in charge of plotting the results). Thus, a model of 1000 connected neurons has 27000 atomic blocks.

After producing and propagating an event, the simulation engine must find which is the block that provokes the next event and when that event is being produced. As PowerDEVS was not originally designed to simulate large networks, it simply looks at the next event time of each block and then it takes the one having the smallest (in case two or more blocks share the minimum time, PowerDEVS uses a priority list to decide among them).

This solution, although being appropriate for small models, is completely inefficient for a large network, as it implies searching along the whole structure at each step. Thus, the cost of each search for the next event time grows linearly with the size of the system.

Taking into account that the number of neuron firings grows linearly with the size of the network, so will grow the number of simulation events. If the search for the next event time also grows linearly, then the total cost will grow at least quadratically with the network dimension. Consequently, we shall obtain something similar to that conventional numerical methods. Thus, if we want to improve this, we need to avoid performing a linear search for the next event time.

The problem of optimizing the search for the next event time in large models has been previously discussed in the literature.

The DEVS abstract simulator of [39] proposes a tree structure for the DEVS model. That way, the search for the next event time is no longer linear. However, in a SNN model, we can have connections from any neuron to any other neuron and there is no natural way of splitting the model in a tree–like form at the modeling stage. The best we can do to reduce the size of the higher hierarchy level is to build a single coupled model for each neuron. However, the number of neurons is still very large and the cost of performing a linear search among them is still linear.

In order to improve the performance of the time scheduling procedure some authors have proposed to parallelize part of the algorithms and also to reduce the search space among a subset of atomic models having a minimum time advance [24, 25]. As we want to show an algorithm working in a single processor, we cannot take the first solution. The second one, although it would improve the performance, will not avoid the linear growth.

Another solution consists in leaving aside from the search to those atomic models in passive states, i.e., with time advance equal to infinite [13, 27]. As before, this solution in a SNN network will not prevent the search for the next event time from growing linearly.

Finally, to accelerate the search among the models with finite time advance, a heap structure is usually implemented [13, 27].

As this solution reduces the cost from linear to logarithmic, we made our implementation based on that. We could have also combined this idea with the other solutions analyzed above. That way, we might have improved further the results. However, we only wanted to reduce the time search from linear to logarithmic in order to prevent the total simulation cost from growing quadratically with the size of the network.

We organized the atomic models with a binary tree structure. Every leaf of the tree contains a reference to an atomic model and to its next event time. Every node of the tree contains the minimum time of its children and the reference to the corresponding atomic model. Thus, the main node of the tree contains the minimum time of the network and the reference to the corresponding atomic model.

After an event, for each atomic model that changed its next event time, we only need to compare it with the time of its brother. If the smaller time does not change, nothing else has to be done. Otherwise, we need to propagate the new minimum time to its father node that will in turn compare it back with its brother node. In the worst case, the propagation will reach the main node. In that worst case, only about $\log_2(N)$ comparisons are performed (where N is the number of blocks) for each atomic model that changed its time advance.

As the number of connections per neuron is bounded by a constant in these models, each event is propagated to a maximum number of blocks which does not depend on the total number of blocks. Thus, the number of comparisons needed after each transition is bounded in a logarithmic way.

According to some preliminary results, the usage of the binary tree allowed us to reduce the total simulation time by two orders of magnitude in the simulation of a model with 1000 neurons.

4 Simulation Results

This section shows and discusses the simulation results. Here, we compare the results obtained with the QSS3 method (in PowerDEVS) using constant and logarithmic quantization, with those obtained using Runge–Kutta–Fehlberg and Bulirsch–Stoer (for both methods we used the code provided in [30] with the addition of discontinuity handling routines).

We worked on three examples: a single isolated neuron, a large network of isolated neurons and finally a large network of interconnected neurons.

4.1 Simulation of a Single Neuron

We first simulated a single neuron with a final simulation time of 1 second. We used the model of Eq.(3). For the parameters, we followed [14] choosing $a = 0.02$, $b = 0.2$, $c = -65 + 15 \cdot r^2$, and $d = 8 - 6 \cdot r^2$, where r is a random variable with uniform distribution in the interval $[0, 1]$. Similarly, $I(t)$ was taken as a constant function with random value.

We ran each simulation 100 times in order to obtain meaningful statistical data. We measured, for different tolerance and quantum settings, the mean absolute error and the total simulation time. We used as reference solution that obtained with QSS3 using a quantum $\Delta Q = 10^{-11}$.

We used this solution as a reference since the analytical solution cannot be obtained. In order to check that the different methods converge to the reference solution, we included simulations performed with those methods using small tolerances.

Figure 7 shows the state trajectories for a particular simulation run. They coincide with the typical solutions of Izhikevich's model.

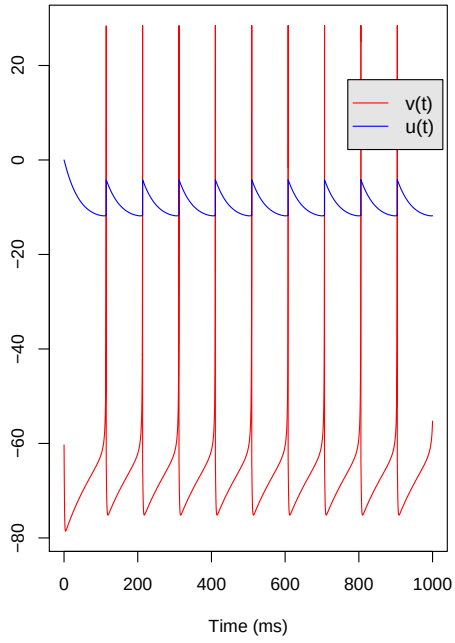

Figure 7: State trajectories of a single neuron simulation using QSS3. The difference between the solutions given by the three numerical integration methods cannot be distinguished by the naked eye.

Tables 1, 2, 3, and 4 summarize the results. Figures 8 and 9 illustrate the evolution of the CPU Time and Error as a function of the tolerance or quantum parameter for the different methods.

For small tolerances, we see that the error of BS and RK45 (i.e., the difference with the reference solution of QSS3) is around $1E-8$. This means that the reference solution is useful up to this tolerance.

Parameter	Mean CPU Time in sec. (Variance)	Mean Error (Variance)
10^{-03}	7.21E-04 (1.69E-04)	6.51E-01 (4.17E-01)
10^{-04}	7.93E-04 (1.76E-04)	9.95E-02 (6.91E-02)
10^{-05}	8.85E-04 (1.93E-04)	6.62E-03 (7.19E-03)
10^{-06}	1.05E-03 (2.09E-04)	3.78E-04 (5.43E-04)
10^{-07}	1.32E-03 (2.41E-04)	4.71E-05 (5.71E-05)
10^{-08}	1.78E-03 (3.00E-04)	4.96E-06 (6.07E-06)
10^{-09}	2.26E-03 (3.42E-04)	6.31E-07 (5.35E-07)
10^{-10}	3.22E-03 (4.63E-04)	8.13E-08 (4.43E-08)
10^{-11}	4.80E-03 (6.50E-04)	7.16E-09 (7.76E-09)

Table 1: Simulation with RK45. Parameter = relative tolerance.

Parameter	Mean CPU Time in sec.(Variance)	Mean Error (Variance)
10^{-03}	1.12E-03 (2.41E-04)	1.41E+00 (6.84E-01)
10^{-04}	1.44E-03 (3.03E-04)	3.67E-01 (2.31E-01)
10^{-05}	1.53E-03 (4.59E-04)	1.40E-01 (9.88E-02)
10^{-06}	2.17E-03 (4.40E-04)	1.05E-02 (1.69E-02)
10^{-07}	2.23E-03 (4.93E-04)	1.70E-04 (2.30E-04)
10^{-08}	2.72E-03 (6.53E-04)	1.53E-05 (1.11E-05)
10^{-09}	2.88E-03 (5.71E-04)	1.26E-06 (1.04E-06)
10^{-10}	3.32E-03 (7.19E-04)	1.57E-05 (8.49E-05)
10^{-11}	3.66E-03 (7.38E-04)	1.24E-08 (1.31E-08)

Table 2: Simulation with BS. Parameter = relative tolerance.

The results exhibit a clear advantage of the RK and BS methods over QSS3. The dynamics do not exhibit many discontinuities and the system is fully interconnected. Thus, QSS3 does not have much to offer here. Moreover, the PowerDEVS implementation (Fig.5) is quite inefficient as every event of the integrators is propagated to several static atomic blocks, which are in turn propagated back along the coupled structure. So, the simulation engine spends more time handling the coupling structure than doing useful calculations. Additionally, the PowerDEVS engine performs some initialization procedures before each simulation that take some fixed time (it takes about 0.01 seconds), so there is a lower bound for the total simulation time.

When it comes to errors and tolerance, we found that the performance of QSS3 compared with RK and BS worsens when the tolerance becomes more

Parameter	Mean CPU Time in sec.(Variance)	Mean Error (Variance)
10^{-03}	2.35E-02 (1.53E-03)	1.16E-01 (7.50E-02)
10^{-04}	3.17E-02 (3.29E-03)	1.74E-02 (9.74E-03)
10^{-05}	5.05E-02 (6.64E-03)	1.52E-03 (2.11E-03)
10^{-06}	8.88E-02 (1.42E-02)	2.64E-04 (1.04E-03)
10^{-07}	1.74E-01 (3.11E-02)	7.74E-06 (2.94E-06)
10^{-08}	3.58E-01 (6.71E-02)	7.63E-07 (3.12E-07)
10^{-09}	7.55E-01 (1.53E-01)	8.26E-08 (5.21E-08)
10^{-10}	1.62E+00 (3.16E-01)	6.71E-09 (6.37E-09)
10^{-11}	3.47E+00 (6.32E-01)	0.00E+00 (0.00E+00)

Table 3: Simulation with QSS3. Parameter = ΔQ.

Parameter	Mean CPU Time in sec.(Variance)	Mean Error (Variance)
10^{-03}	1.92E-02 (8.78E-04)	1.15E+00 (3.08E-01)
10^{-04}	2.26E-02 (3.74E-03)	1.66E-01 (1.10E-01)
10^{-05}	2.82E-02 (2.97E-03)	6.73E-03 (8.38E-03)
10^{-06}	4.12E-02 (5.60E-03)	1.41E-03 (3.55E-03)
10^{-07}	6.96E-02 (1.08E-02)	4.67E-04 (1.39E-03)
10^{-08}	1.31E-01 (2.31E-02)	1.89E-04 (8.97E-04)
10^{-09}	2.66E-01 (5.33E-02)	3.02E-06 (5.27E-06)
10^{-10}	5.51E-01 (1.09E-01)	3.02E-07 (5.46E-07)
10^{-11}	1.18E+00 (2.50E-01)	1.89E-08 (2.52E-08)

Table 4: Simulation with logarithmic QSS3. Parameter = ΔQ_{rel}.

stringent (Fig.8). This is due to the fact that QSS3 is only a third order accurate algorithm and thus the CPU Time grows with the cubic root of the required accuracy.

4.2 Simulation of Networks of Isolated Neurons

In this second experiment, we simulated different systems of disconnected neurons. We ran different simulations of systems with 10, 100, 1000, and 2000 neurons using the same methods of the previous example. In order to have similar errors with the different algorithms, we selected a tolerance of 10^{-4} for RK, and 10^{-5} for BS. Similarly, we selected a quantum $\Delta Q = 10^{-3}$ for QSS3 and $\Delta Q_{rel} = 10^{-4}$ for QSS3 with logarithmic quantization. With these values, according to the simulation results of the previous example, all the methods have a mean absolute error around 10^{-1}.

Table 5 exhibits the mean time (after 30 simulation runs) required by each method to complete the simulation. The final simulation time was set to 1 second.

Figure 10 plots the CPU time required by each method for the different number of neurons.

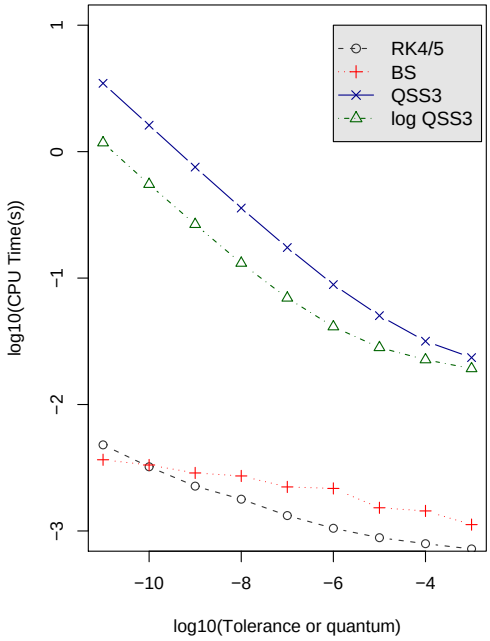

Figure 8: CPU Time vs. tolerance (quantum) for different methods.

Method	10 neurons	100 neurons
RK	1.41E-02 (1.78E-02)	1.31E-01 (3.00E-02)
BS	1.30E-02 (5.27E-03)	3.70E-01 (8.33E-02)
QSS3	2.08E-01 (2.02E-01)	7.70E-01 (2.19E-01)
QSS3 log	1.24E-01 (8.39E-02)	5.93E-01 (9.40E-02)
Method	1000 neurons	2000 neurons
RK	1.04E+01 (8.36E-01)	3.88E+01 (3.30E+00)
BS	3.71E+01 (4.33E+00)	1.35E+02 (1.53E+01)
QSS3	7.89E+00 (2.09E+00)	1.72E+01 (3.49E+00)
QSS3 log	7.07E+00 (1.74E+00)	1.61E+01 (3.51E+00)

Table 5: Mean CPU Time in sec. for simulating 10, 100, 1000, and 2000 disconnected neurons. The variance is reported between parentheses.

The results become now more interesting. For small systems (10 to 100 neurons), QSS3 methods show a poor performance compared with BS and RK. However, when the number of neurons increases, QSS3 rapidly outperforms both discrete time algorithms.

This fact can be easily explained taking into account the way in which QSS

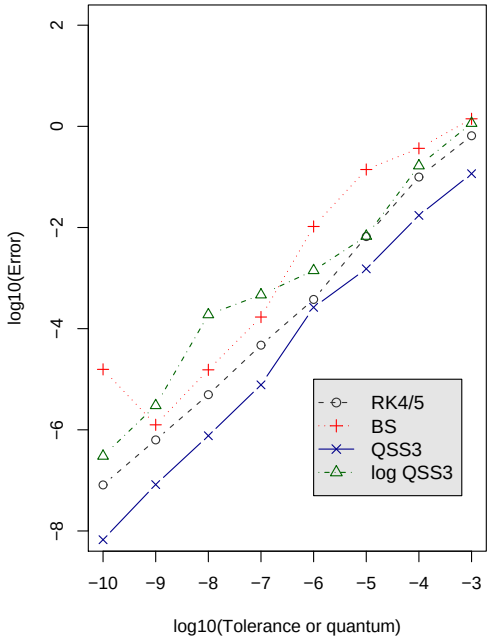

Figure 9: Error vs. tolerance (quantum) for different methods.

methods exploit activity. As we mentioned in Section 2.2, discrete time algorithms must evaluate all the state derivatives at each step, and they must restart the whole simulation after each discontinuity (i.e., after each firing). As the number of firings as well as the size of the right hand side function of the ODE grow linearly with the size of the network, the discrete time algorithms experience a quadratic growth of the CPU time.

Although the QSS3 method is inefficient to simulate a single neuron, every step and every firing only provokes calculations at one neuron. Thus, the algorithm only performs computations where the changes occur, i.e., where the system shows some activity. In consequence, the CPU Time grows almost linearly with the number of neurons. This fact can be observed in Table 5.

4.3 Simulation of Networks of Interconnected Neurons

In this last test we simulated different networks of interconnected neurons. To this end, we followed the scheme proposed in the Benchmark I of [5].

We considered networks composed by 400, 1000, 2000, and 4000 neurons. In each case, 80% of the neurons are of excitatory type while the remaining 20% are of inhibitory type. We also established that each neuron is connected (in a

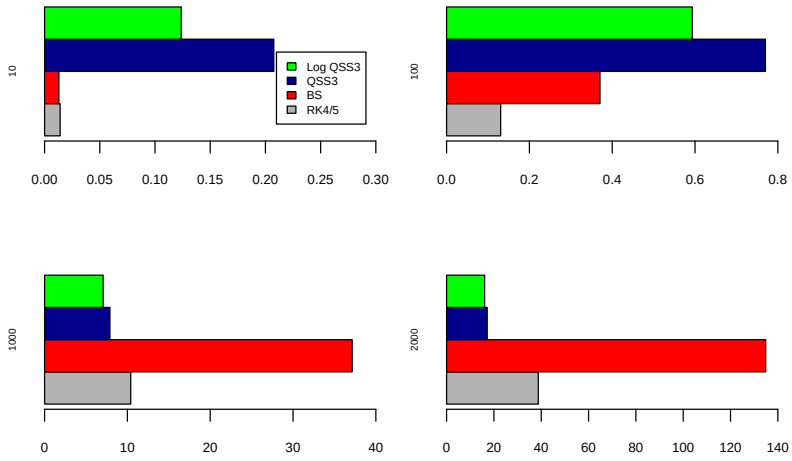

Figure 10: CPU Time (sec.) for RK, BS, QSS3, and QSS3 logarithmic in the simulation of a system of 10, 100, 1000, and 2000 disconnected neurons.

random way) with 80 neurons.

We compare the performance of the four algorithms of the previous example, with the same tolerance and quantum settings, but now we selected a final simulation time of 250 milliseconds. As before, we ran each simulation 30 times and computed the mean CPU time required for each algorithm. Table 6 summarizes the results.

Method	400	1000	2000	4000
RK	9.79 (1.62)	68.89 (7.94)	300.29 (25.84)	1231.24 (76.98)
BS	30.08 (3.69)	165.58 (16.40)	568.40 (43.50)	2124.14 (293.23)
QSS3	17.74 (1.59)	61.33 (2.71)	139.76 (4.79)	301.76 (11.00)
QSS3 log	12.56 (0.85)	44.15 (2.19)	101.80 (3.55)	218.19 (7.61)

Table 6: Mean CPU Time (sec.) for 400, 1000, 2000, and 4000 interconnected neurons. The variance is reported between parentheses.

Figure 11 plots the CPU Time as a function of the number of neurons for each algorithm.

These new results agree with the previous case of disconnected neurons. When the number of neurons becomes larger, QSS3 methods become more efficient than both discrete time algorithms.

As we observed in the previous example, Figure 11 shows that the CPU Time grows almost linearly in QSS3 methods, while it grows at least quadratically in the other algorithms.

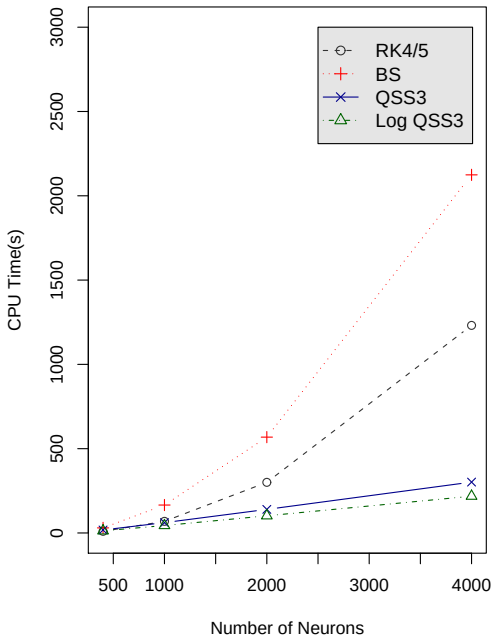

Figure 11: Mean CPU Time vs. Number of Neurons.

5 Conclusions

In this article we studied the use of QSS algorithms in the simulation of spiking neural networks. We found that their efficient discontinuity handling and their activity driven features offer very important advantages over the most widely used discrete time methods.

The greatest advantage exhibited by the QSS methods is that its computational costs grow almost linearly with the number of neurons, while it grows at least quadratically for discrete time methods. Consequently, the performance of the QSS algorithms in the simulation of large networks of neurons is clearly superior.

As a side result, we implemented an algorithm to manage the time advance of the simulation engine based on a binary tree of the submodels. While the cost for computing the next event time is linear with the number of submodels in most DEVS implementations, in our algorithm it results only logarithmic.

Although the results are promising and the QSS methods noticeably improve the performance of discrete time algorithms in the systems analyzed, some work has to be done before claiming that this discrete event approach constitutes a valid and general alternative for spiking neural networks simulation.

First, we need to perform experiments with different models (so far, we used only Izhikevich's model with a particular type of synapses). We conjecture that we shall find the same advantages, but we cannot affirm that without running simulations.

In the current work, as we explained in Section 3, the models were built using an ad–hoc C++ program that generated the PowerDEVS model structure. If we want to convince people from the neurosciences community to use our algorithms, we definitely need to develop a better end–user interface.

Moreover, it is possible that PowerDEVS (or any other DEVS–based modeling tool) is not the best choice to implement these simulations. Any general purpose DEVS simulation engine performs several tasks that are not useful in the context of simulating a QSS approximation of an ODE. We are currently working on the development of standalone QSS solvers. Some preliminary results show a simulation speedup of one order of magnitude with respect to the same QSS approximation executed by PowerDEVS. Thus, if we can develop a specific end–user interface for modeling large SNN and then we integrate them with these standalone solvers, we can expect a significant improvement of the results shown in this article.

Finally, for larger networks of neurons, we will need to tackle the problem of parallelization of the algorithms.

6 Acknowledgments

We wish to acknowledge support by CONICET under grant PIP–2009/2011–00183.

References

[1] T.P. Abbott and L.F. Vogels. Signal propagation and logic gating in networks of integrate-and-fire neurons. *Journal of Neuroscience*, 25(46):10786–10795, 2005.

[2] T. Beltrame. Design and development of a dymola/modelica library for discrete event-oriented systems using devs methodology. Master's thesis, ETH Zurich, Zurich, Switzerland, 2006.

[3] F. Bergero and E. Kofman. PowerDEVS. A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 2010. in Press.

[4] Y. Boiko and G. Wainer. Modeling of neural decoder based on binary spiking neurons in devs. In *Proceedings of the 2009 Spring Simulation Multiconference*, San Diego, California, 2009.

[5] R. Brette et al. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience*, 23(3):349–398, 2007.

[6] F. Cellier, E. Kofman, G. Migoni, and M. Bortolotto. Quantized state system simulation. In *Proceedings of SummerSim 08 (2008 Summer Simulation Multiconference)*, Edinburg, Scotland, 2008.

[7] F.E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, New York, 2006.

[8] M. D'Abreu and G. Wainer. M/cd++: modeling continuous systems using modelica and devs. In *Proceedings of MASCOTS 2005*, Atlanta, GA, 2005.

[9] B. Ermentrout. Type i membranes, phase resetting curves, and synchrony. *Neural Computation*, 8(5):979–1001, 1996.

[10] E. Hairer, S. Norsett, and G. Wanner. *Solving Ordinary Differential Equations I.* Springer, Berlin, 1991.

[11] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems.* Springer, Berlin, 1991.

[12] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117(4):500–544, August 1952.

[13] X. Hu and B. Zeigler. A high performance simulation engine for large-scale cellular devs models. In *Proceedings of the 2004 High Performance Computing Symposium (HPC'04), Advanced Simulation Technologies Conference*, Arlington, VA, 2004.

[14] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, November 2003.

[15] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactiond on Neural Networks*, 15(5):1063–1070, September 2004.

[16] E. M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. The MIT Press, 2007.

[17] Rajanikanth Jammalamadaka. Activity Characterization of Spatial Models: Application to Discrete Event Solution of Partial Differential Equations. Master's thesis, The University of Arizona, 2003.

[18] E. Kofman. A Second Order Approximation for DEVS Simulation of Continuous Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 78(2):76–89, 2002.

[19] E. Kofman. Discrete Event Simulation of Hybrid Systems. *SIAM Journal on Scientific Computing*, 25(5):1771–1797, 2004.

[20] E. Kofman. A Third Order Discrete Event Simulation Method for Continuous System Simulation. *Latin American Applied Research*, 36(2):101–108, 2006.

[21] E. Kofman. Relative Error Control in Quantization Based Integration. *Latin American Applied Research*, 39(3):231–238, 2009.

[22] E. Kofman and S. Junco. Quantized State Systems. A DEVS Approach for Continuous System Simulation. *Transactions of SCS*, 18(3):123–132, 2001.

[23] L. Lapicque. Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarization. *J. Physiol. Pathol. Gen.*, 9:620–635, 1907.

[24] Q. Liu and G. Wainer. Accelerating large-scale devs-based simulation on the cell processor. In *Proceedings of the 2010 Spring Simulation Multiconference: Symposium on Theory of Modeling and Simulation. DEVS Integrative M&S Symposium (DEVS 2010)*, Orlando, Florida, 2010.

[25] Q. Liu, G. Wainer, L. Lu, and M. Perrone. Novel performance optimization of large-scale discrete-event simulation on the cell broadband engine. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation (HPCS 2010)*, Caen, France, 2010.

[26] A. Muzy, A. Aiello, P.A. Santoni, B.P. Zeigler, J.J. Nutaro, and R. Jammalamadaka. Discrete event simulation of large-scale spatial continuous systems. In *Proceedings of IEEE Intenational Conference on Systems, Man and Cybernetics*, volume 4, pages 2991 – 2998, 2005.

[27] A. Muzy and J. Nutaro. Algorithms for efficient implementations of the devs & dsdevs abstract simulators. In *1st Open International Conference on Modeling & Simulation*, pages 401–407, ISIMA / Blaise Pascal University, France, 2005.

[28] A. Muzy, J. Nutaro, B.P. Zeigler, and P. Coquillard. Modeling and simulation of fire spreading through the activity tracking paradigm. *Ecological Modelling*, 219(1-2):212–225, 2008.

[29] J. Nutaro and B. Zeigler. On the stability and performance of discrete event methods for simulating continuous systems. *Journal of Computational Physics*, 227(1):797–819, 2007.

[30] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes in c: The art of scientific computing. second edition, 1992.

[31] G. Quesnel, R. Duboz, E. Ramat, and M. Traoré. Vle: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Computer Simulation Conference*, pages 367–374, San Diego, California, 2007.

[32] Mayrhofer R., M. Affenzeller, H. Prähofer, G. Hfer, and A. Fried. DEVS simulation of spiking neural networks. In *Proceedings of Cybernetics and Systems (EMCSR)*, volume 2, pages 573–578, 2002.

[33] G. D. Smith, C. L. Cox, S. M. Sherman, and J. Rinzel. Fourier analysis of sinusoidally driven thalamocortical relay neurons and a minimal integrate-and-fire-or-burst model. *Journal of Neurophysiology*, 83(1):588–610, 2000.

[34] R. B. Stein. Some models of neuronal variability. *Biophysical Journal*, 7:37–68, 1967.

[35] R. D. Stewart and W. Bair. Spiking neural network simulation: numerical integration with the parker-sochacki method. *Journal of Computational Neuroscience*, 27(1):115–133, August 2009.

[36] G. Wainer. Performance analysis of continuous cell-devs models. In *Proceedings of High Performance Computing & Simulation Conference (HPC&S); 18th European Simulation Multiconference*, Magdeburg, Germany, 2004.

[37] G. Wainer and N. Giambiasi. Cell-devs/gdevs for complex continuous systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 81(2):137–151, 2005.

[38] B. Zeigler. Devs representation of dynamical systems: Event-based intelligent control. *Proceedings of the IEEE*, 77(1):72–80, 1989.

[39] B. Zeigler, T.G. Kim, and H. Praehofer. *Theory of Modeling and Simulation. Second edition.* Academic Press, New York, 2000.

[40] B. Zeigler and J.S. Lee. Theory of quantized systems: formal basis for DEVS/HLA distributed simulation environment. In *SPIE Proceedings*, pages 49–58, 1998.

[41] G. Zheng, A. Tonnelier, and D. Martinez. Voltage-stepping schemes for the simulation of spiking neural networks. *Journal of Computational Neuroscience*, 26(3):409–423, 2009.