Vectorised algorithms for spiking neural network simulation

Romain Brette[1,2] and Dan F. M. Goodman[1,2]

[1] Laboratoire Psychologie de la Perception, CNRS and Université Paris Descartes, Paris, France

[2] Département d'Etudes Cognitives, Ecole Normale Supérieure, Paris, France

October 20, 2010

Abstract

High-level languages (Matlab, Python) are popular in neuroscience because they are flexible and accelerate development. However, for simulating spiking neural networks, the cost of interpretation is a bottleneck. We describe a set of algorithms to simulate large spiking neural networks efficiently with high-level languages using vector-based operations. These algorithms constitute the core of Brian, a spiking neural network simulator written in the Python language. Vectorised simulation makes it possible to combine the flexibility of high-level languages with the computational efficiency usually associated with compiled languages.

Keywords: simulation, algorithms, spiking neural networks, integrate-and-fire, vectorisation

1 Introduction

Computational modelling has become a popular approach to understand the nervous system and link experimental findings at different scales, from ion channels to behaviour. Although there are several mature neural network simulators (Brette et al., 2007), which can simulate complex biophysical models, including Neuron (Carnevale and Hines, 2006) and Genesis (Bower and Beeman, 1998), or very large-scale spiking models, for example NEST (Gewaltig and Diesmann, 2007), many scientists prefer to rely on their own simulation code, often written in Matlab. One reason might be that custom code allows more flexible development and control over the models, while in compiled simulators, using a non-standard model involves writing an extension, which is not straightforward (for example, in NMODL for Neuron (Hines and Carnevale, 2000), or in C for NEST).

However, implementing simulations using hand-written code rather than dedicated simulation software has several disadvantages. First of all, code written by individual scientists is more likely to contain bugs than simulators which have been intensively worked on and tested. This may lead to difficulties in reproducing the results of such simulations (Djurfeldt et al., 2007; Schutter, 2008; Cannon et al., 2007). In addition, hand-written code can be more difficult to understand, which may slow the rate of scientific progress. Several initiatives are underway to develop standardised markup languages for simulations in computational neuroscience (Goddard et al., 2001; Garny et al., 2008; Morse, 2007), in order to address issues of reproducibility. However, these do not address the issue of flexibility, which has driven many scientists to write their own code. To address this issue, we developed a simulator written in Python (Goodman and Brette, 2009), which is a high-level interpreted language with dynamic typing. This allows the user to define their own custom model in the script and easily interact with the simulation, while benefiting from a variety of libraries for scientific computing and visualisation that are freely available. All of the major simulation packages now include Python interfaces (Eppler et al., 2008; Hines et al., 2009) and the PyNN project (Davison et al., 2008) is working towards providing a unified interface to them, but these are only interfaces to the compiled code of the simulators, and therefore do not solve the flexibility problem. In the Brian simulator, the core of simulations is written in Python. This allows the user to easily define new models, but the obvious problem with this choice is that programs usually run much slower with an interpreted language than with a compiled language. To address this problem, we developed vectorisation algorithms which minimize the number of interpreted operations, so that the performance approaches that of compiled simulators.

Vectorising code is the strategy of minimising the amount of time spent in interpreted code compared to highly optimised array functions, which operate on a large number of values in a single operation. For example, the loop `for i in range(0,N): z[i]=x[i]+y[i]` can be vectorised as `z=x+y`, where x, y and z are vectors of length N. In the original code, the number of interpreted operations is proportional to N. In the vectorised code, there are only two interpreted operations and two vectorised ones. Thus, if N is large, interpretation takes a negligible amount of time, compared to compiled vector-based operations. In Python with the Numpy scientific package, the operation above is about 40 times faster when it is vectorised for $N = 1000$. In clock-driven simulations, all neurons are synchronously updated at every tick of the clock. If all models have the same functional form (but possibly different parameter values), then the same operations are executed many times with different values. Thus, these operations can be easily vectorised and if the network contains many neurons, then the interpretation overhead becomes negligible.

Neuron models can usually be described as differential equations and discrete events (spikes), so that the simulation time can be divided into the cost of updating neuron states and of propagating spikes, summarised in the following formula (Brette et al., 2007):

$$\begin{array}{ccc} \textit{Update} & + & \textit{Propagation} \\ c_U \times \dfrac{N}{\mathrm{d}t} & + & c_P \times F \times N \times p \end{array}$$

where c_U is the cost of one update and c_P is the cost of one spike propagation, N is the number of neurons, p is the number of synapses per neuron, F is the average firing rate and $\mathrm{d}t$ is the time step (the cost is for one second of biological time). The principle of vectorisation is to balance each interpreted operation with a vector-based operation that acts on many values, so that the proportion of time spent in interpretation vanishes as the number of neurons or synapses increases. In general, this amounts to replacing all loops by vector-based operations, but this is not a systematic rule: a loop can be kept if each operation inside the loop is vectorised. For example, if update is vectorised over neurons and propagation is vectorised over synapses (i.e., the action of one outgoing spike on all target neurons is executed in one vector-based operation), then the simulation cost is:

$$\begin{array}{ccc} \textit{Update} & + & \textit{Propagation} \\ \dfrac{c_U \times N + c_I}{\mathrm{d}t} & + & F \times N \times (c_P \times p + c_I) \end{array}$$

where c_I is the interpretation cost. If both N and p are large, then interpretation does not degrade simulation speed.

We will start by presenting data structures that are appropriate for vectorised algorithms (section 2). We will then show how to build a network in a vectorised way (section 3), before presenting our vectorised algorithms for neural simulation (section 4). These include state updates, threshold and reset, spike propagation, delays and synaptic plasticity (section 5). Similar data structures and algorithms were already used in previous simulators (Brette et al., 2007), but not in the context of vectorisation.

In this paper, we will illustrate the structures and algorithms with code written in Python using the Numpy library for vector-based operations, but most algorithms can also be adapted to Matlab or Scilab. The algorithms are implemented in the Brian simulator, which can be downloaded from `http://briansimulator.org`. It is open source and freely available under the CeCILL license (compatible with the GPL license). Appendix B shows the Python code for a complete example simulation, which is independent of the Brian simulator.

2 Models and data structures

2.1 Vectorised containers

We use several combinations of some basic data structures designed for easy vectorisation. These structures are illustrated in Figure 1.

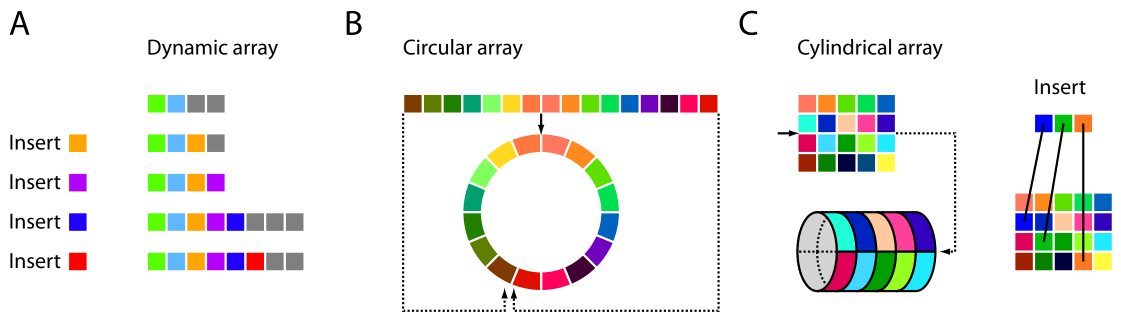

Figure 1: Basic structures. A *dynamic array* is an array with unused space (grey squares), which is resized by a fixed factor (here, 2) when it is full. A *circular array* is implemented as an array with a cursor indicating the current position. A *cylindrical array* is implemented as a two-dimensional array with a cursor indicating the current position. Insertion can be easily vectorised.

2.1.1 Dynamic arrays

A dynamic array is a standard data structure (used, for example, in most implementations of the C++ STL `vector` class, and for Python's `list` class) that allows for both fast indexing and resizing. It is a linear array with some possibly unused space at the end. When we want to append a new element to the dynamic array, if there is unused space we place the element there, otherwise we resize the array to make space for the new elements. In detail, it is an array `val` of length n in which the first $m \leq n$ elements are used and the remainder are free. When an element is appended to the dynamic array, if $m < n$ then the new element is simply placed in `val[m]` and m is increased by 1. If $m = n$ (the array is full), then the array is first resized to an array of length αn (for some constant $\alpha > 1$) at cost $O(n)$. Thus, every time the array is resized, the operation costs $O(n)$ and leaves space for $(\alpha - 1)n$ elements. Therefore the average cost per extra element is $O(n/(\alpha-1)n) = O(1)$ (this is sometimes referred to as the *amortized* cost per element).

2.1.2 Circular arrays

A circular array `x` is an array which wraps around at the end, and which can be rotated at essentially no cost. It is implemented as an array `y` of size `N` and a `cursor` variable indicating the current origin of the array. Indexing `x` is circular in the sense that `x[i+N]=x[i]=y[(i+cursor)%N]`, where % is the modulus operator, giving the remainder after division. Rotating the array corresponds to simply changing the `cursor` variable. Circular arrays are convenient for storing data that only needs to be kept for a limited amount of time. New data is inserted at the origin, and the array is then rotated. After several such insertions, the data will be rotated back to the origin and overwritten by new data. The benefits of this type of array for vectorisation are that insertion and indexing are inexpensive, and it does not involve frequent allocations and deallocations of memory (which would cause memory fragmentation and cache misses as data locality would be lost).

2.1.3 Cylindrical arrays

A cylindrical array `X` is a two-dimensional version of a circular array. It has the property that `X[t+M, i] = X[t, i]` where `i` is a non-circular index (typically a neuron index in our case), `t` is a circular index (typically representing time), and `M` is the size of the circular dimension. In the same way as for circular arrays, it is implemented as a 2D array `Y` with a `cursor` for the first index, where the `cursor` points to the origin. Cylindrical arrays are then essentially circular arrays in which an array can be stored instead of a single value. The most important property of cylindrical arrays for vectorisation is that multiple elements of data can be stored at different indices in a vectorised way. For example, if we have an array `V` of values, and we want to store each element of `V` in a different position in the circular array, this can be done in a vectorised way as follows. To perform n assignments `X[t_k, i_k]=v_k` ($k \in \{1, \ldots, n\}$), we can execute a single vectorised operation: `X[T, I]=V`, where `T`$=(t_1, \ldots, t_n)$, `I`$=(i_1, \ldots, i_n)$ and `V`$=(v_1, \ldots, v_n)$ are three vectors. For the underlying array, it means `Y[(T+cursor)%M, I]=V`.

2.2 The state matrix

Neuron models can generally be expressed as differential equations and discrete events (Protopapas et al. 1998; Brette et al. 2007; Goodman and Brette 2008; and for a recent review of the history of the idea, Plesser and Diesmann 2009):

$$\begin{aligned}\frac{\mathrm{d}\mathbf{X}}{\mathrm{d}t} &= f(\mathbf{X}) \\ \mathbf{X} &\leftarrow g_i(\mathbf{X}) \qquad \text{upon spike from synapse i}\end{aligned}$$

where $\mathbf{X}$ is a vector describing the state of the neuron (e.g. membrane potential, excitatory and inhibitory conductance...). This formulation encompasses integrate-and-fire models but also multicompartmental models with synaptic dynamics. Spikes are emitted when some threshold condition is satisfied: $\mathbf{X} \in \mathbf{A}$, for instance $V_m \geq \theta$ for integrate-and-fire models or $\mathrm{d}V_m/\mathrm{d}t \geq \theta$ for Hodgkin-Huxley type models (where $V_m = X_0$, the first coordinate of $\mathbf{X}$). Spikes are propagated to target neurons, possibly with a transmission delay, where they modify the value of a state variable ($\mathbf{X} \leftarrow \mathbf{g_i}(\mathbf{X})$). For integrate-and-fire models, the membrane potential is reset when a spike is produced: $\mathbf{X} \leftarrow r(\mathbf{X})$.

Some models are expressed in an integrated form, as a linear superposition of postsynaptic potentials (PSPs), for example the spike response model (Gerstner and Kistler, 2002):

$$V(t) = \sum_i w_i \sum_{t_i} PSP(t - t_i) + V_{\text{rest}}$$

where $V(t)$ is the membrane potential, V_{rest} is the rest potential, w_i is the synaptic weight of synapse i, and t_i are the timings of the spikes coming from synapse i. This model can be rewritten as a hybrid system, by expressing the PSP as the solution of a linear differential system. For example, consider an α-function: $PSP(t) = (t/\tau)e^{1-t/\tau}$. This PSP is the integrated formulation of the associated kinetic model: $\tau\dot{g} = y - g$, $\tau\dot{y} = -y$, where presynaptic spikes act on y (that is, cause an instantaneous increase in y), so that the model can be reformulated as follows:

$$\begin{aligned}\tau\frac{\mathrm{d}V}{\mathrm{d}t} &= V_{\text{rest}} - V + g \\ \tau\frac{\mathrm{d}g}{\mathrm{d}t} &= -g \\ g &\leftarrow g + w_i \qquad \text{upon spike from synapse i}\end{aligned}$$

Virtually all post-synaptic potentials or currents described in the literature (e.g. bi-exponential functions) can be expressed this way. Several authors have described the transformation from phenomenological expressions to the hybrid system formalism for synaptic conductances and currents (Destexhe et al., 1994b,a; Rotter and Diesmann, 1999; Giugliano, 2000), short-term synaptic depression (Tsodyks and Markram, 1997; Giugliano et al., 1999), and spike-timing-dependent plasticity (Song et al., 2000) (see also section 5). A systematic method to derive the equivalent differential system for a PSP (or postsynaptic current or conductance) is to use the Laplace transform or the Z-transform (Köhn and Wörgötter, 1998) (see also Sanchez-Montanez 2001), by seeing the PSP as the impulse response of a linear time-invariant system (which can be seen as a filter (Jahnke et al., 1999)).

It follows from the hybrid system formulation that the state of a neuron can be stored in a vector $\mathbf{X}$. In general, the dimension of this vector is at least the number of synapses. However, in many models used in the literature, synapses of the same type share the same linear dynamics, so that they can be reduced to a single set of differential equations per synapse type, where all spikes coming from synapses of the same type act on the same variable (Lytton, 1996; Song et al., 2000; Morrison et al., 2005; Brette et al., 2007), as in the previous example. In this case, which we will focus on, the dimension of the state vector does not depend on the number of synapses but only on the number of synapse types, which is assumed to be small (e.g. excitatory and inhibitory). Consider a group of N neurons with the same m state variables and the same dynamics (the same differential equations). Then updating the states of these neurons involves repeating the same operations N times, which can be vectorised. We will then define a *state matrix* $\mathbf{S}$ for this group made of p rows of length N, each row defining the values of the same state variable for all N neurons. We choose this matrix orientation (rather than N rows of length p) because the vector describing one state variable for the whole group is stored as a contiguous array in memory, which is more efficient for vector-based operations. It is still possible to have heterogeneous groups in this way, if parameter values (the membrane time constant for example) are different between neurons. In this case, parameters can be treated as state variables and included in the state matrix.

The state of all neurons in a model is then defined by a set of state matrices, one for each neuron type, where the type is defined by the set of differential equations, threshold conditions and resets. Each state matrix has a small number of rows (the number of state variables) and a large number of columns (the number of neurons). Optimal efficiency is achieved with the smallest number of state matrices. Thus, neurons should be gathered in groups according to their models rather than functionally (e.g. layers).

2.3 Connectivity structures

2.3.1 Structures for network construction and spike propagation

When a neuron spikes, the states of all target neurons are modified (possibly after a delay), usually by adding a number to one state variable (the synaptic weight). Thus, the connectivity structure must store, for each neuron, the list of target neurons, synaptic weights and possibly transmission delays. Other variables could be stored, such as transmission probability, but these would involve the same sort of structures.

Suppose we want to describe the connectivity of two groups of neurons P and Q (possibly identical or overlapping), where neurons from P project to neurons in Q. Because the structure should be appropriate for vectorised operations, we assume that each group is homogeneous (same model, but possibly different parameters) and presynaptic spikes from P all act on the same state variable in all neurons in Q. These groups could belong to larger groups. When a spike is emitted, the program needs to retrieve the list of target neurons and their corresponding synaptic weights. This is most efficient if these lists are stored as contiguous vectors in memory.

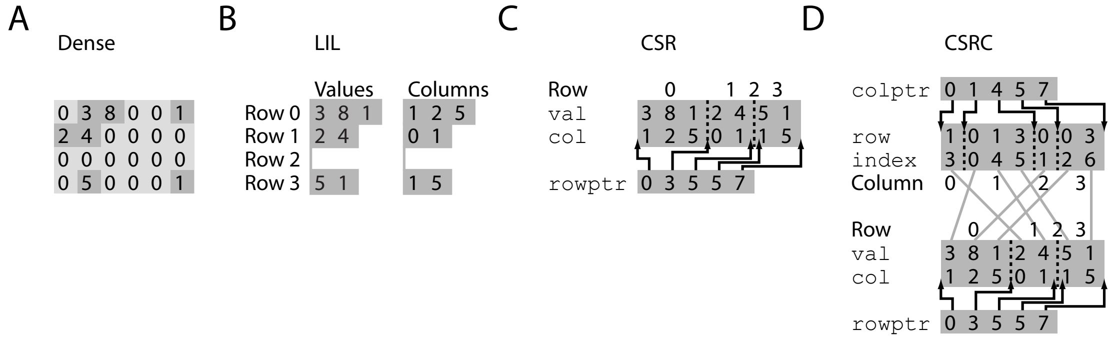

Figure 2: Matrix structures for synaptic connectivity. In all matrices, the number of rows is the size of the presynaptic neuron group and the number of columns is the size of the postsynaptic neuron group. In *dense matrices*, all synaptic weights are explicitly stored, where zero values signal absent or silent synapses. In the *list of lists* (LIL) format, only non-zero elements are stored (i.e., it is a sparse format). It consists of a list of rows, where each row is a list of column indices of non-zero elements (target neurons) and a list of their corresponding values (synaptic weights). The *compressed sparse row* (CSR) format is a similar sparse matrix format, but the list of rows are concatenated into two arrays containing all values (`val`) and all column indices (`col`). The position of each row in these arrays is indicated in an array `rowptr`. The *compressed sparse row and column* (CSRC) format extends the CSR format for direct column access. The array `row` is the transposed version of `col`: it indicates the row indices of non-zero elements for each column. The array `index` contains the position of each non-zero element in the `val` array, for each column.

The simplest case is when these groups are densely connected together. Synaptic weights can then be stored in a (dense) matrix $\mathbf{W}$, where $W_{i,j}$ is the synaptic weight between neuron i in P and neuron j in Q. Here we are assuming that a given neuron does not make multiple synaptic contacts with the same postsynaptic neuron — if it does, several matrices could be used. Thus, the synaptic weights for all target neurons of a neuron in P are stored as a row of **W**, which is contiguous in memory. Delays can be stored in a similar matrix. If connectivity is sparse, then a dense matrix is not efficient. Instead, a sparse matrix should be used (Figure 2). Because we need to retrieve rows when spikes are produced, the appropriate format is a list of rows, where each row is defined by two vectors: one defining the list of target neurons (as indices in Q) and one defining the list of synaptic weights for these neurons. When constructing the network, these are best implemented as chained lists (list of lists or LIL format), so that one can easily insert new synapses. But for spike propagation, it is more efficient to implement this structure with fixed arrays, following the CSR format (Compressed Sparse Row). The standard CSR matrix consists of three arrays `val`, `col` and `rowptr`. The non-zero values of the matrix are stored in the array `val` row by row, that is, starting with all nonzero values in row 0, then in row 1, and so on. This array has length `nnz`, the number of nonzero elements in the matrix. The array `col` has the same structure, but contains the column indices of all non-zero values, that is, the indices of the postsynaptic neurons. Finally, the array `rowptr` contains the position for each row in the arrays `val` and `col`, so that row i is stored in `val[j]` at indices j with `rowptr[i]` $\leq j <$ `rowptr[i+1]`. The memory requirement for this structure, assuming values are stored as 64 bit floats and indices are stored as 32 bit integers, is 12`nnz` bytes. This compressed format can be converted from the list format just before running the simulation (this takes about the same time as construction for 10 synapses per neuron, and much less for denser networks).

As for the neuron groups, there should be as few connection matrices as possible to minimise the number of interpreted operations. Thus, synapses should be gathered according to synapse type (which state variable is modified) and neuron type (pre- and postsynaptic neuron groups) rather than functionally. Connectivity between subgroups of the same type (layers for example) should be defined using submatrices (with array views).

2.3.2 Structures for spike-timing-dependent plasticity

In models of spike-timing-dependent plasticity (STDP), both presynaptic spikes and postsynaptic spikes act on synaptic variables. In particular, when a postsynaptic spike is produced, the synaptic weights of all presynaptic neurons are modified, which means that a column of the connection matrix must be accessed, both for reading and writing. This is straightforward if this matrix is dense, but more difficult if it is sparse and stored as a list of rows.

For fast column access, we augment the standard CSR matrix described above with column information to form what we will call a compressed sparse row and column (CSRC) matrix (Figure 2). We add three additional arrays `index`, `row` and `colptr` to the CSR matrix. These play essentially the same roles as `val`, `col` and `rowptr`, except that (a) they are column oriented rather than row oriented, as in a compressed sparse column (CSC) matrix, and (b) instead of storing values in `val`, we store the indices of the corresponding values in `index`. So column j consists of all elements with row indices `row[i]` for `colptr[j]` $\leq i <$ `colptr[j+1]` and values `val[index[i]]`. This allows us to perform vectorised operations on columns in the same way as on rows. However, the memory requirements are now 20`nnz` bytes, and column access will be slower than row access because of the indirection through the `index` array, and because the data layout is not contiguous (which will reduce cache efficiency). The memory requirements could be reduced to as low as 10`nnz` bytes by using 32 bit floats and 16 bit indices (as long as the number of rows and columns were both less than $2^{16} = 65536$).

2.4 Structures for storing spike times

In order to implement delays and refractoriness, we need to have a structure to store the indices of neurons which have spiked recently (Figure 3). In order to minimise memory usage, this structure should only retain these indices for a fixed period (the maximum delay or refractory period). The standard structure for this sort of behavior is a circular array (see 2.1.2), which has been previously used for this purpose in non-vectorised simulations (Morrison et al., 2005; Izhikevich, 2006). Each time step, we insert the indices of all neurons that spiked in the group: `x[0:len(spikes)]=spikes`, and shift the cursor by `len(spikes)`. After insertion the most recent spikes are always in `x[i]` for negative `i`. To extract the spikes produced at a given timestep in the past, we need to store a list of the locations of the spikes stored at these timesteps in the underlying array `y`. This is also implemented in a circular array, of length `duration` where `duration` is the number of timesteps we want to remember spikes for. With this array we can insert new spikes and extract the set of indices of neurons that spiked at any given time (within `duration`) at a cost of only $O(n)$ where n is the number of spikes returned. The algorithm is vectorised over spikes. The length of the circular array `x` depends *a priori* on the number of spikes produced, which is unknown before the simulation. Thus we use a dynamic array (see 2.1.1) to resize the spike container if necessary. As in the case of the standard dynamic array, this makes insertion an amortized $O(1)$ operation.

The Python code for this structure is shown in Appendix A.

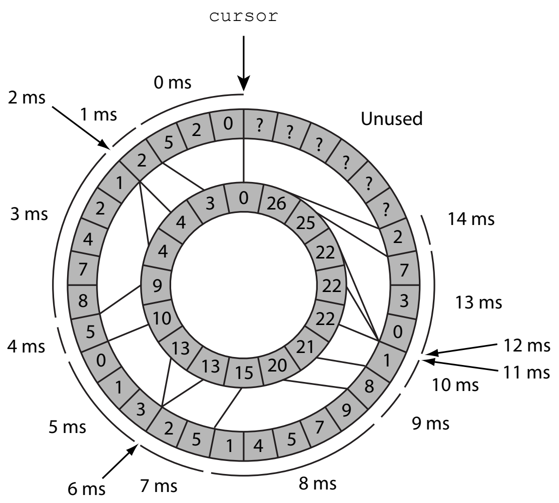

Figure 3: Spike container. A circular dynamic array `x` (outer ring) stores the indices of all neurons that spiked in previous time steps, where the cursor indicates the current time step. Another circular array (inner ring) stores the location of previous time steps in the dynamic array.

3 Construction

Building and initialising the network can be time consuming. In the introduction, we presented a formula for the computational cost of a neural simulation. This expression can be mirrored for network construction by calculating the cost of initialising N neurons and constructing connection matrices with Np entries:

$$\begin{array}{ccc} \textit{Neurons} & + & \textit{Connections} \\ c_N \times N & + & c_C \times N \times p \end{array}$$

where c_N is the cost of initialising one neuron and c_C is the cost of initialising one synapse. The cost of the second term can be quite significant, because it is comparable to the cost of simulating spike propagation for a duration $1/F$, where F is the average firing rate. Vectorising neuron initialisation is straightforward, because the values of a variable for all neurons is represented by a row of the state matrix. For example, the following Python instruction initialises the second state variable of all neurons to a random value between -70 and -55: `S[1,:]=-70+rand(N)*15`. Vectorising the construction of connection matrices is more complex. A simple option is to vectorise over synapses. As an example, the following Brian instruction connects all pairs of neurons from the same group P with a distance-dependent weight (the topology is a ring):

```
C.connect_full(P, P, weight=lambda i, j:cos((2*pi/N)*(i-j)))
```

where C is the connection object, which contains the dense connection matrix `C.W`. When this method is called, the matrix is built row by row by calling the `weight` function with arguments i and j (in Python, the `lambda` keyword defines a function), where i is the row index and j is the vector of indices of target neurons. This is equivalent to the following Python code (using the Numpy package for vector-based operations):

```
j = arange(N)
for i in range(N):
  C.W[i, :] = cos((2*pi/N)*(i-j))
```

These instructions now involve only $O(N)$ interpreted operations. More generally, by vectorising over synapses, the analysis of the computation cost including interpretation reads:

$$\begin{array}{ccc} \textit{Neurons} & + & \textit{Connections} \\ c_N \times N + c_I & + & N(c_C \times p + c_I) \end{array}$$

where c_I is the interpretation cost, and the interpretation overhead is negligible when both N and p are large.

With sparse matrices, both the synaptic weights and the list of target neurons must be initialised. A typical case is that of random connectivity, where any given pair of neurons has a fixed probability x of being connected and synaptic weights are constant. Again, the connection matrix can be initialised row by row as follows. Suppose the presynaptic group has N neurons and the postsynaptic group has M neurons. For each row of the connection matrix (i.e., each presynaptic neuron), pick the number k of target neurons at random according to a binomial distribution with parameters N and x. Then pick k indices at random between 0 and $M-1$. These indices are the list of target neurons, and the list of synaptic weights is a constant vector of length k. With this algorithm, constructing each row involves a constant number of vector-based operations. In Python, the algorithm reads:

```
W_target = []
W_weight = []
for i in range(N):
  k = binomial(M, x, 1)[0]
  target = sample(xrange(M), k)
  target.sort()
  W_target.append(target)
  W_weight.append([w]*k)
```

where `[w]*k` is a list of k elements `[w, w ,..., w]`. Clearly, synaptic weights need not be constant and their construction can also be vectorised as previously shown.

4 Simulation

Simulating a network for one timestep consists in the following: 1) updating the state variables, 2) checking threshold conditions and resetting neurons, 3) propagating spikes. This process is illustrated in Figure 4. We now describe vectorisation algorithms for each of these steps, leaving synaptic plasticity to the next section (section 5).

4.1 State updates

Between spikes, neuron states evolve continuously according to a set of differential equations. Updating the states consists in calculating the value of the state matrix at the next time step $\mathbf{S}(t+\mathrm{d}t)$, given the current value $\mathbf{S}(t)$. Vectorising these operations is straightforward, as we show in the next paragraphs. Figure 5 shows that with these vectorised algorithms, the interpretation time indeed vanishes for large neuron groups.

4.1.1 Linear equations

A special case is when the differential system is linear. In this case, the state vector of a neuron at time $t+\mathrm{d}t$ depends linearly on its state vector at time t. Therefore the following equality holds: $\mathbf{X}(t+\mathrm{d}t) = \mathbf{AX}(t) + \mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are determined by the differential system and can be calculated at the start of the simulation (Hirsch and Smale, 1974; Rotter and Diesmann, 1999; Morrison et al., 2007).

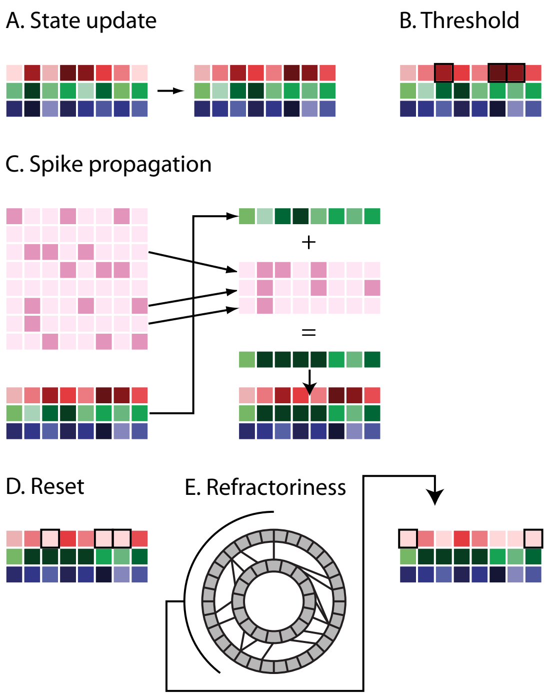

Figure 4: Simulation loop. For each neuron group, the state matrix is updated according to the model differential equations. The threshold condition is tested, which gives a list of neurons that spiked in this time step. Spikes are then propagated, using the connection matrix (pink). For all spiking neurons, the weights of all target synapses (one row vector per presynaptic neuron, pink rows) are added to the row of the target state matrix that corresponds to the modified state variable (green row). Neurons that just spiked are then reset, along with neurons that spiked in previous time steps, within the refractory period.

Each state vector is a column of the state matrix $\mathbf{S}$. It follows that the update operation for all neurons in the group can be compactly written as a matrix product: $\mathbf{S}(t + \mathrm{d}t) = \mathbf{A}\mathbf{S}(t) + \mathbf{B}\mathbf{J}^T$, where $\mathbf{J}$ is a vector of ones (with N elements).

4.1.2 Nonlinear equations

When equations are not linear, numerical integration schemes must be used, such as Euler or Runge-Kutta. Although these updates cannot be written as a matrix product, vectorisation is straightforward: it only requires us to substitute state vectors (rows of matrix $\mathbf{S}$) to state variables in the update scheme. For example, the Euler update $x(t + \mathrm{d}t) = x(t) + f(x(t))\mathrm{d}t$ is replaced by $\mathbf{X}(t + \mathrm{d}t) = \mathbf{X}(t) + f(\mathbf{X}(t))\mathrm{d}t$, where the function f now acts element-wise on the vector $\mathbf{X}$.

4.1.3 Stochastic equations

Stochastic differential equations can be vectorised similarly, by substituting vectors of random values to random values in the stochastic update scheme. For example, a stochastic Euler update would read: $\mathbf{X}(t + \mathrm{d}t) = \mathbf{X}(t) + f(\mathbf{X}(t))\mathrm{d}t + \sigma\mathbf{N}(t)\sqrt{\mathrm{d}t}$, where $\mathbf{N}(t)$ is a vector of normally distributed random numbers. There is no specific difficulty in vectorising stochastic integration methods.

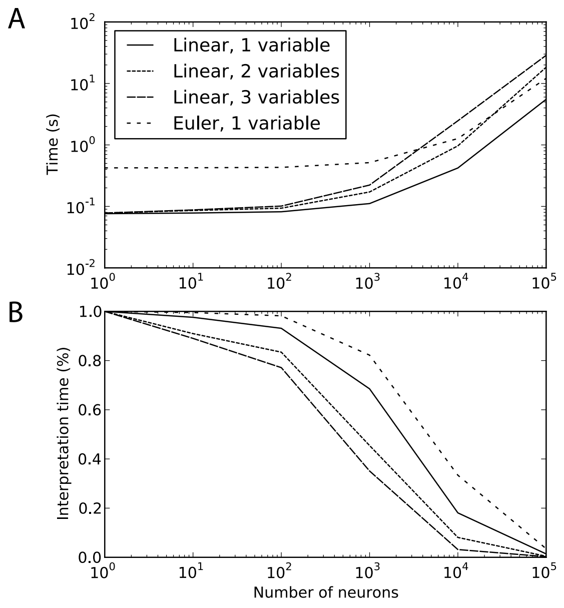

Figure 5: Interpretation time for state updates. A. Simulation time vs. number of neurons for vectorised state updates, with linear update and Euler integration (log scale). B. Proportion of time spent in interpretation vs. number of neurons for the same operations. Interpretation time was calculated as the simulation time with no neurons.

4.2 Threshold, reset and refractoriness

4.2.1 Threshold

Spikes are produced when a threshold condition is met, defined on the state variables: $\mathbf{X} \in \mathbf{A}$. A typical condition would be $X_0 \geq \theta$ for an integrate-and-fire model (where $V_m = X_0$), or $X_0 \geq X_1$ for a model with adaptive threshold (where X_1 is the threshold variable) (Deneve, 2008; Platkiewicz and Brette, 2010). Vectorising this operation is straightforward: 1) apply a boolean vector operation on the corresponding rows of the state matrix, e.g. `S[0, :]>theta` or `S[0, :]>S[1, :]`, 2) transform the boolean vector into an array of indices of true elements, which is also a generic vector-based operation. This would be typically written as `spikes=find(S[0, :]>theta)`, which outputs the list of all neurons that must spike in this timestep.

4.2.2 Reset

In the same way as the threshold operation, the reset consists of a number of operations on the state variables ($\mathbf{X} \leftarrow r(\mathbf{X})$), for all neurons that spiked in the timestep. This implies: 1) selecting a submatrix of the state matrix where columns are selected according to the `spikes` vector (indices of spiking neurons), and 2) applying the reset operations on rows of this submatrix. For example, the vectorised code for a standard integrate-and-fire model would read: `S[0, spikes]=Vr`, where `Vr` is the reset value. For an adaptive threshold model, the threshold would additionally increase by a fixed amount: `S[1, spikes]+=a`.

4.2.3 Refractoriness

Absolute refractoriness in spiking neuron models generally consists in clamping the membrane potential at a reset value V_r for a predefined duration. To implement this mechanism, we need to extract the indices of all neurons that have spiked in the last k timesteps. These indices are stored contiguously in the circular array defined in section 2.4. To extract this subvector `refneurons`, we only need to obtain the starting index in the circular array where the spikes produced in the k^{th} previous timestep were stored. Resetting then amounts to executing the following vectorised code:

`S[0, refneurons]=Vr`.

This implicitly assumes that all neurons have the same refractory period. For heterogeneous refractoriness, we can store an array `refractime` of length the number of neurons, which gives the next time at which the neuron will stop being refractory. If the refractory periods are given as an array `refrac`, and at time `t` neurons with indices in the array `spikes` fire, we update `refractime[spikes]=t+refrac[spikes]`. At any time, the indices of the refractory neurons are given by the boolean array `refneurons=refractime>t`. Heterogeneous refractory times are more computationally expensive than homogeneous ones because the latter operation requires N comparisons (for N the number of neurons), rather than just picking out directly the indices of the neurons that have spiked (on average $N \times F \times \mathrm{d}t$ indices, where F is the average firing rate). A mixed solution that is often very efficient in practice (although no better in the worst case) is to test the condition `refneurons=refractime>t` only for those neurons that fired within the most recent period `max(refrac)` (which may include repeated indices). If the array of these indices is called `candidates`, then the code reads: `refneurons=candidates[refractime[candidates]>t]`. The mixed solution can be chosen when the size of the array `candidates` is smaller than the number of neurons (which is not always true if neurons can fire several times within the maximum refractory period).

4.3 Propagation of spikes

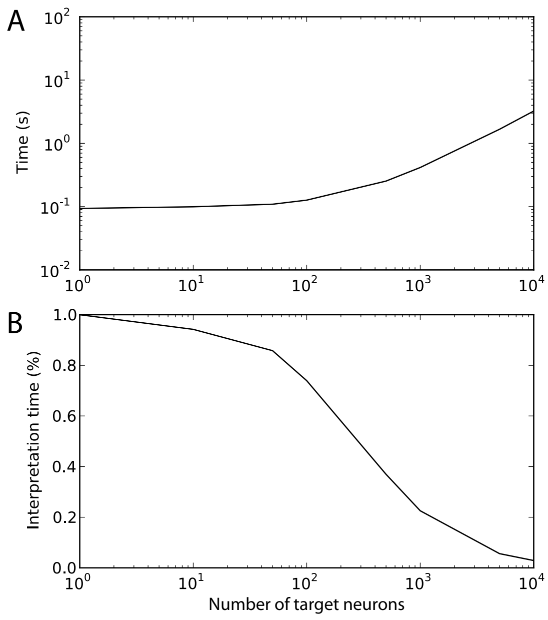

Figure 6: Interpretation time for spike propagation. We measured the computation time to propagate one spike to a variable number of target neurons, using vector-based operations. A. Simulation time vs. number of target neurons per spike for spike propagation, calculated over 10000 repetitions. B. Proportion of time spent in interpretation vs. number of target neurons per spike for the same operations.

Here we consider the propagation of spikes with no transmission delays, which are treated in the next section (4.4). Consider synaptic connections between two groups of neurons where presynaptic spikes act on postsynaptic state variable k. In a non-vectorised way, the spike propagation algorithm can be written as follows:

```
for i in spikes:
  for j in postsynaptic[i]:
    S[k, j] += W[i, j]
```

where the last line can be replaced by any other operation, and `spikes` is the list of indices of neurons that spike in the current timestep (`postsynaptic` is a list of postsynaptic neurons with no repetition). The inner loop can be vectorised in the following way:

```
for i in spikes:
  S[k, postsynaptic[i]] += W[i, postsynaptic[i]]
```

Although this code is not fully vectorised since a loop remains, it is vectorised over postsynaptic neurons. Thus, if the number of synapses per neuron is large, the interpretation overhead becomes negligible. For dense connection matrices, the instruction in the loop is simply `S[k, :]+=W[i, :]`. For sparse matrices, this would be implemented as `S[k, row_target[i]]+=row_val[i]`, where `row_target[i]` is the array of indices of neurons that are postsynaptic to neuron i and `row_val[i]` is the array of non-zero values in row i of the weight matrix.

Figure 6 shows that, as expected, the interpretation in this algorithm is negligible when the number of synapses per neuron is large.

4.4 Transmission delays

4.4.1 Homogeneous delays

Implementing spike propagation with homogeneous delays between two neuron groups is straightforward using the structure defined in section 2.4. The same algorithms as described above can be implemented, with the only difference that vectorised state modifications are iterated over spikes in a previous timestep (extracted from the spike container) rather than in the current timestep.

4.4.2 Heterogeneous delays

Heterogeneous delays cannot be simulated in the same way. Morrison et al. (2005) introduced the idea of using a queue for state modifications on the postsynaptic side, implemented as a cylindrical array (see 2.1.3). For an entire neuron group, this can be vectorised by combining these queues to form a cylindrical array `E[t, i]` where `i` is a neuron index and `t` is a time index. Figure 7 illustrates how this cylindrical array can be used for propagating spikes with delays. At each time step, the row of the cylindrical array corresponding to the current timestep is added to the state variables of the target neurons: `S[k, :]+=E[0, :]`. Then, the values in the cylindrical array at the current timestep are reset to 0 (`E[0, :] = 0`) so that they can be reused, and the `cursor` variable is incremented. Emitted spikes are inserted at positions in the cylindrical array that correspond to the transmission delays, that is, if the connection from neuron i to j has transmission delay `d[i, j]`, then the synaptic weight is added to the cylindrical array at position `E[d[i, j], j]`. This addition can be vectorised over the postsynaptic neurons j as described in section 2.1.3.

5 Synaptic plasticity

5.1 Short term plasticity

In models of short term synaptic plasticity, synaptic variables (e.g. available synaptic resources) depend on presynaptic activity but not on postsynaptic activity. It follows that, if the parameter values are the same for all synapses, then it is enough to simulate the plasticity model for each neuron in the presynaptic group, rather than for each synapse (Figure 7).

To be more specific, we describe the implementation of the classical model of short-term plasticity of Markram and Tsodyks (Markram et al., 1998). This model can be described as a hybrid system (Tsodyks and Markram, 1997; Tsodyks et al., 1998; Loebel and Tsodyks, 2002; Mongillo et al., 2008; Morrison et al., 2008), where two synaptic variables x and u are governed by the following differential equations:

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= (1-x)/\tau_d \\ \frac{\mathrm{d}u}{\mathrm{d}t} &= (U-u)/\tau_f\end{aligned}$$

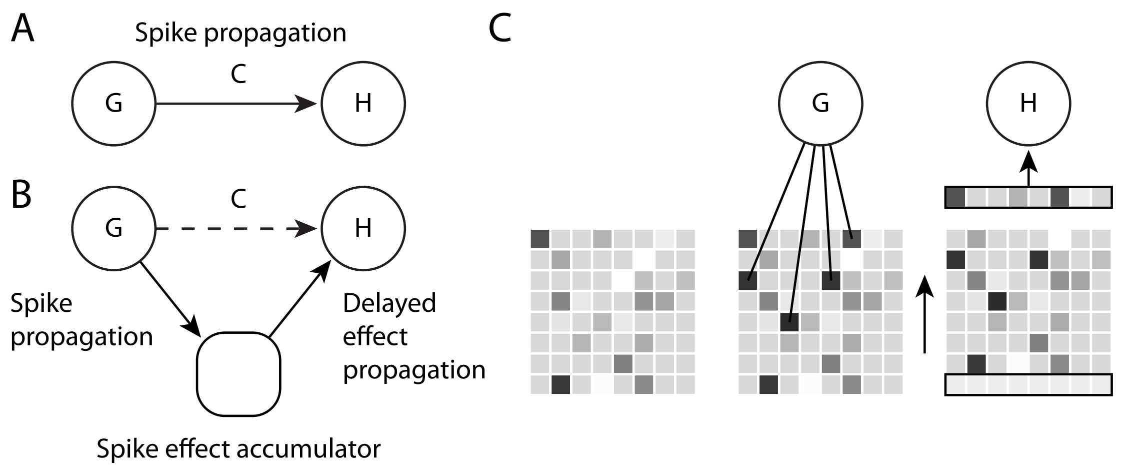

Figure 7: Heterogeneous delays. A. A connection C is defined between neuron groups G and H. B. With heterogenous delays, spikes are immediately transmitted to a spike effect accumulator, then progressively transmitted to neuron group H. C. The accumulator is a cylindrical array which contains the state modifications to be added to the target group at each time step in the future. For each presynaptic spike in neuron group G, the weights of target synapses are added to the accumulator matrix at relevant positions in time (vertical) and target neuron index (horizontal). The values in the current time step are then added to the state matrix of H and the cylindrical array is rotated.

where τ_d, τ_f are time constants and U is a parameter in $[0,1]$. Each presynaptic spike triggers modifications of the variables:

$$\begin{aligned} x &\leftarrow x(1-u) \\ u &\leftarrow u+U(1-u) \end{aligned}$$

Synaptic weights are multiplicatively modulated by the product ux (before update). Variables x and u are in principle synapse-specific. For example, x corresponds to the fraction of available resources at that synapse. However, if all parameter values are identical for all synapses, then these synaptic variables always have the same value for all synapses that share the same presynaptic neuron, since the same updates are performed. Therefore, it is sufficient to simulate the synaptic model for all presynaptic neurons.

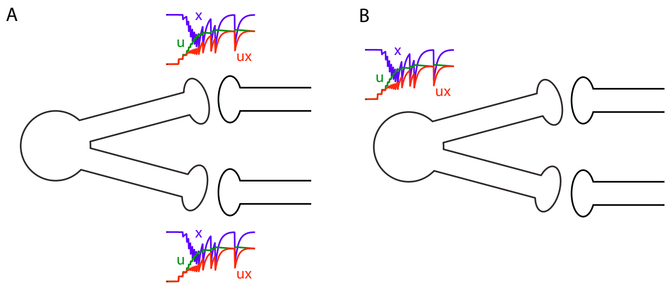

Figure 8: Short-term plasticity. A. Synaptic variables x and u are modified at each synapse with presynaptic activity, according to a differential model. B. If the model parameters are identical, then the values of these variables are identical for all synapses that share the same presynaptic neuron, so that only one model per presynaptic neuron needs to be simulated.

We note that the differential equations are equivalent to the state update of a neuron group with variables x and u, while the discrete events are equivalent to a reset. Therefore, the synaptic model can be simulated as a neuron group, using the vectorised algorithms that we previously described.

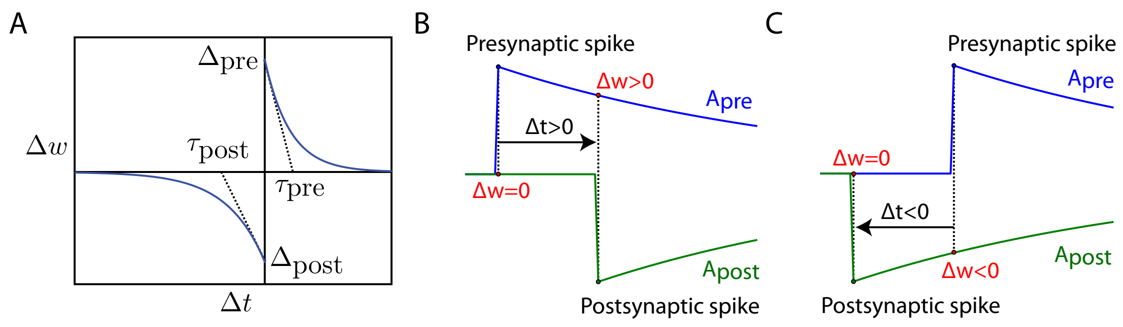

Figure 9: Spike-Timing-Dependent Plasticity (STDP). A. Synaptic modification Δw as a function of the timing difference Δt between post- and presynaptic spikes. In this example, STDP modification is piecewise exponential. B. When a presynaptic spike arrives, variable A_{pre} (presynaptic trace) is increased and then decays exponentially. The synaptic weight also increases by the value of A_{post} (postsynaptic trace) ($\Delta w = 0$ in this case). When a postsynaptic spike arrives, variable A_{post} is decreased and the synaptic weight is increased by the value of A_{pre} ($\Delta w > 0$ here). C. When these spikes occur in the reverse order, the synaptic weight decreases.

The only change is that spike propagation must be modified so that synaptic weights are multiplicatively modulated by ux. This simply amounts to replacing the operation `S[k,:]+=W[i,:]` by `S[k,:]+=W[i,:]*u[i]*x[i]`. State updates could be further accelerated with event-driven updates, since their values are only required at spike times, as described in (Markram et al., 1998) and implemented e.g. in NEST (Morrison et al., 2008).

However, if the parameter values of the synaptic model (τ_d,τ_f,U) are heterogeneous, or if spike transmission is probabilistic and the updates of x and/or u only occur at successful transmissions, then the values of the synaptic variables are different even when they share the same presynaptic neuron. In this case, we must store and update as many variables as synapses. Updates could still be vectorised (e.g. over synapses) but the simulation cost is much higher.

5.2 Spike-timing-dependent plasticity

A spike-timing-dependent plasticity (STDP) model typically consists of 1) a set of synaptic variables, 2) a set of differential equations for these variables and 3) discrete updates of these variables and/or of the synaptic weight at the times of presynaptic and postsynaptic spikes. Very often, STDP models are defined by the weight modification for a pair of pre- and postsynaptic spikes as a function of their timing difference, where all pairs may interact or only neighboring spikes. However, these models can always be expressed in the hybrid system form described above (Morrison et al., 2008). For example, in the classical pair-based exponential STDP model (Song et al., 2000), synaptic weights are modified depending on the timing difference between pre- and postsynaptic spikes $\Delta t = t_{\text{post}} - t_{\text{pre}}$, by an amount

$$\Delta w = f(\Delta t) = \begin{cases} \Delta_{\text{pre}} e^{-\Delta t/\tau_{\text{pre}}} & \text{if } \Delta t > 0 \\ \Delta_{\text{post}} e^{\Delta t/\tau_{\text{post}}} & \text{if } \Delta t < 0 \end{cases} \tag{1}$$

The modifications induced by all spike pairs are summed. This can be written as a hybrid system by defining two variables A_{pre} and A_{post}, which are “traces” of pre- and postsynaptic activity and are governed by the following differential equations (see Figure 9):

$$\begin{aligned} \tau_{\text{pre}} \frac{\mathrm{d}}{\mathrm{d}t} A_{\text{pre}} &= -A_{\text{pre}} \\ \tau_{\text{post}} \frac{\mathrm{d}}{\mathrm{d}t} A_{\text{post}} &= -A_{\text{post}} \end{aligned}$$

When a presynaptic spike occurs, the presynaptic trace is updated and the weight is modified according to the postsynaptic trace:

$$\begin{aligned} A_{\text{pre}} &\rightarrow A_{\text{pre}} + \Delta_{\text{pre}} \\ w &\rightarrow w + A_{\text{post}} \end{aligned}$$

When a postsynaptic spike occurs, the symetrical operations are performed:

$$\begin{aligned} A_{\text{post}} &\rightarrow A_{\text{post}} + \Delta_{\text{post}} \\ w &\rightarrow w + A_{\text{pre}} \end{aligned}$$

With this plasticity rule, all spike pairs linearly interact, because the differential equations for the traces are linear and their updates are additive. If only nearest neighboring spikes interact, then the update of the trace A_{pre} should be $A_{\text{pre}} \rightarrow \Delta_{\text{pre}}$ (and similarly for A_{post}).

In the synaptic plasticity model described above, the value of the synaptic variable A_{pre} (resp. A_{post}) only depends on presynaptic (resp. postsynaptic) activity. In this case we shall say that the synaptic rule is *separable*, that is, all synaptic variables can be gathered into two independent pools of presynaptic and postsynaptic variables. Assuming that the synaptic rule is the same for all synapses (same parameter values), this implies that the value of A_{pre} (resp. A_{post}) is identical for all synapses that share the same presynaptic (resp. postsynaptic) neuron. Therefore, in the same way as for short term plasticity, we only need to simulate as many presynaptic (resp. postsynaptic) variables A_{pre} (resp. A_{post}) as presynaptic (resp. postsynaptic) neurons. Simulating the differential equations that define the synaptic rule can thus be done exactly in the same way as for simulating state updates of neuron groups, and the operations on synaptic variables ($A_{\text{pre}} \rightarrow A_{\text{pre}} + \Delta_{\text{pre}}$) can be simulated as resets. However, the modification of synaptic weights requires specific vectorised operations.

When a presynaptic spike is produced by neuron i, the synaptic weights for all postsynaptic neurons j are modified by an amount $A_{\text{post}}[j]$ (remember that there is only one postsynaptic variable per neuron in the postsynaptic group). This means that the entire row i of the weight matrix is modified. If the matrix is dense, then the vectorised code would simply read `W[i, :]+=A_post`, where `A_post` is a vector. Similarly, when a postsynaptic spike is produced, a column of the weight matrix is modified: `W[:, i]+=A_pre`. This is not entirely vectorised since one needs to loop over pre- and postsynaptic spikes, but the interpretation overhead vanishes for a large number of synapses per neuron. If the matrix is sparse, the implementation is slightly different since only the non-zero elements must be modified. If `row_target[i]` is the array of indices of neurons that are postsynaptic to neuron i and `row_val[i]` is the array of non-zero values in row i, then the code should read: `row_val[i]+=A_post[row_target[i]]`. The symetrical operation must be done for postsynaptic spikes, which implies that the weight matrix can be accessed by column (see section 2.3.2).

5.2.1 STDP with delays

When transmission delays are considered, the timing of pre- and postsynaptic spikes in the STDP rule should be defined from the synapse's point of view. If the presynaptic neuron spikes at time t_{pre}, then the spike timing at the synapse is $t_{\text{pre}} + d_a$, where d_a is the axonal propagation delay. If the postsynaptic neuron spikes at time t_{post}, then the spike timing at the synapse is $t_{\text{post}} + d_d$, where d_d is the dendritic backpropagation delay. In principle, both these delays should be specified, in addition to the total propagation delay. The latter may not be the sum $d_a + d_d$ because dendritic backpropagation and forward propagation are not necessary identical. Various possibilities are discussed in Morrison et al. (2008). Presumably, if dendritic backpropagation of action potentials is active (e.g. mediated by sodium channels) while the propagation of postsynaptic potentials is passive, then the former delay should be shorter than the latter one. For this reason, we choose to neglect dendritic backpropagation delays, which amounts to considering that transmission delays are purely axonal. An alternative (non-vectorised) solution requiring that dendritic delays are longer than axonal delays is given in the appendix to Morrison et al. (2007).

Figure 10 shows an example where N presynaptic neurons make excitatory synaptic connections to a single postsynaptic neuron. The delay from presynaptic neuron i to the postsynaptic neuron is $i \times d$. Now suppose each presynaptic neuron fires simultaneously at time 0, and the postsynaptic neuron fires at time $(N/2)d$. At the synapses, presynaptic spikes arrive at times $i \times d$ while the postsynaptic spike occurs at time $(N/2)d$. In this case synapse i would experience a weight modification of Δw for $\Delta t = (N/2)d - id$. More generally, for each pair t_{pre} and t_{post} of pre and postsynaptic firing times, with a delay d, the weight modification is $\Delta w = f(\Delta t)$ for $\Delta t = t_{\text{post}} - (t_{\text{pre}} + d)$.

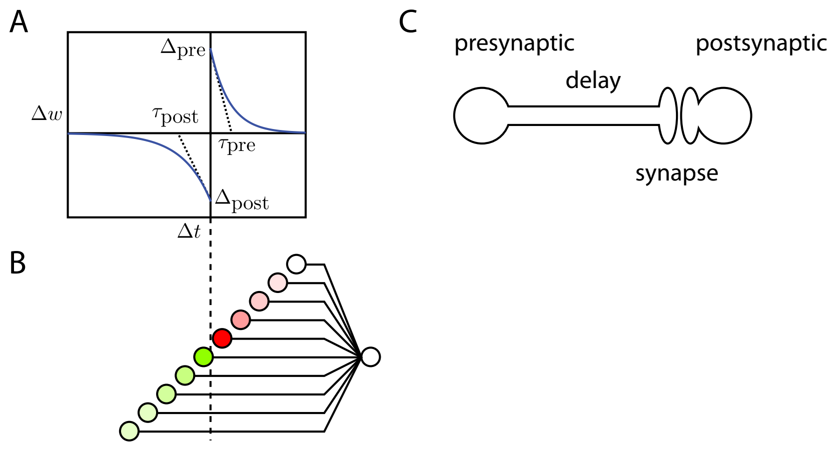

Figure 10: Delays in STDP. We assume that delays are only axonal, rather than dendritic. The timing of pre- and postsynaptic spikes in the synaptic modification rule is seen from the synapse's point of view. Here all N presynaptic neurons fire at time 0 and the postsynaptic neuron fires at time $N/2$. The transmission delay for synapse i is $i \times d$, as reflected by the length of axons. Then synapses from red neurons are potentiated while synapses from green neurons are depressed (color intensity reflects the magnitude of synaptic modification).

In the hybrid system formulation, the discrete events for the traces A_{pre} and A_{post} are unchanged, but the synaptic modifications occur at different times:

$$\begin{aligned} w(t_{\text{post}}) &\rightarrow w(t_{\text{post}}) + A_{\text{pre}}(t_{\text{post}} - d) \\ w(t_{\text{pre}} + d) &\rightarrow w(t_{\text{pre}} + d) + A_{\text{post}}(t_{\text{pre}} + d) \end{aligned}$$

The postsynaptic update (first line) can be easily vectorised over presynaptic neurons (which have different delays d), if the previous values of A_{pre} are stored in a cylindrical array (see 2.1.3). At each time step, the current values of A_{pre} and A_{post} are stored in the row of the array pointed to by a `cursor` variable, which is then incremented. However, the presynaptic update (second line) cannot be directly vectorised over postsynaptic neurons because the weight modifications are not synchronous (each postsynaptic neuron is associated to a specific delay d). We suggest doing these updates at time $t_{\text{pre}} + D$ instead of $t_{\text{pre}} + d$, where D is the maximum delay:

$$w(t_{\text{pre}} + D) \rightarrow w(t_{\text{pre}} + D) + A_{\text{post}}(t_{\text{pre}} + d)$$

The synaptic modification is unchanged (note that we must use $A_{\text{post}}(t_{\text{pre}} + d)$ and not $A_{\text{post}}(t_{\text{pre}} + D)$) but slightly postponed. Although it is not mathematically equivalent, it should make an extremely small difference because individual synaptic modifications are very small. In addition, the exact time of these updates in biological synapses is not known, because synaptic modifications are only tested after many pairings.

6 Discussion

We have presented a set of vectorised algorithms for simulating spiking neural networks. These algorithms cover the whole range of operations that are required to simulate a complete network model, including delays and synaptic plasticity. Neuron models can be expressed as sets of differential equations, describing the evolution of state variables (either neural or synaptic), and discrete events, corresponding to actions performed at spike times. The integration of differential equations can be vectorised over neurons, while discrete events can be vectorised over presynaptic or

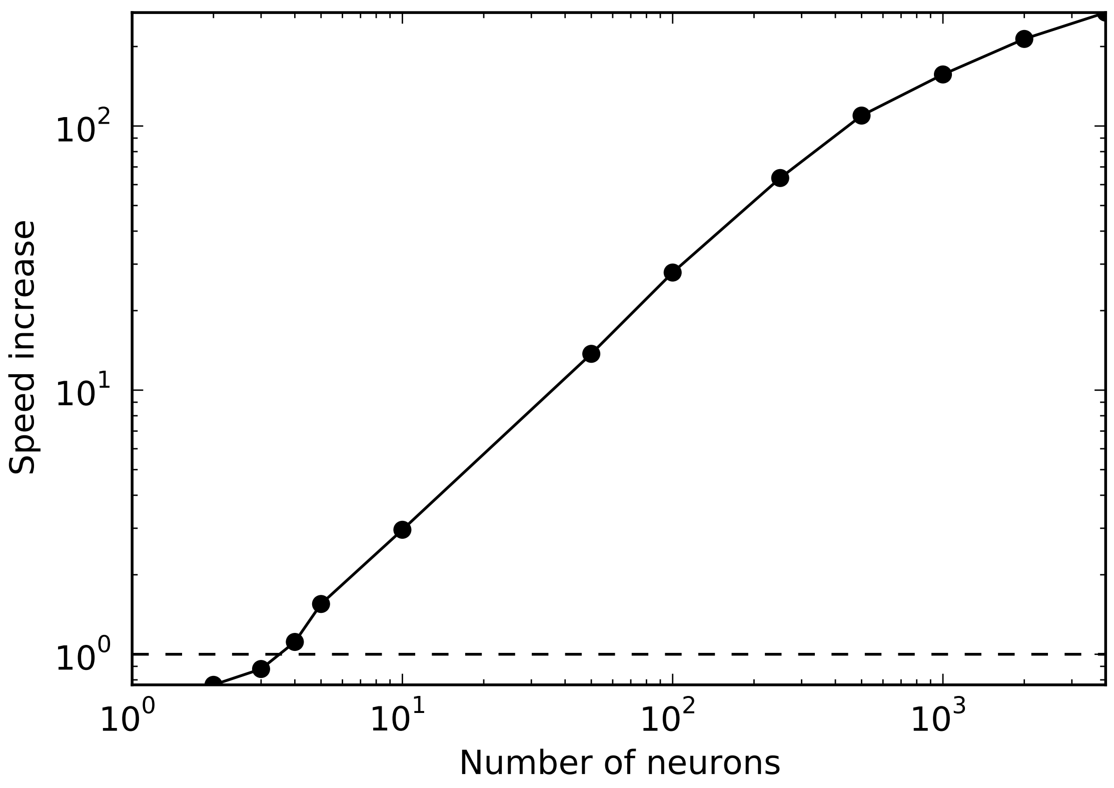

Figure 11: Speed improvement with vectorisation. We simulated the random network described in Appendix B with various numbers of neurons and compared it to a non-vectorised implementation (with loops, also written in Python). Vectorised code is faster for more than 3 neurons.

postsynaptic neurons (i.e., targets of the event under consideration). Thus, if N is the number of neurons and p is the average number of synapses per neuron, then the interpretation overhead vanishes for differential equations (state updates) when N is large (Fig. 5) and for discrete events (spike propagation, plasticity) when p is large (Fig. 6). We previously showed that the Brian simulator, which implements these algorithms, is only about twice as slow as optimized custom C code for large networks (Goodman and Brette, 2008). In Fig. 11, we compared our vectorised implementation of a random network (Appendix B) with a non-vectorised implementation, also written in Python. Vectorised code is faster for more than 3 neurons, and is several hundred times faster for a large number of neurons. In principle, the algorithms could be further optimised by also vectorising discrete events over emitted spikes, but this would probably require some specific vector-based operations (in fact, matrix-based operations).

An interesting extension would be to port these algorithms to graphics processing units (GPUs), which are parallel co-processors available on modern graphics cards. These are inexpensive units designed originally and primarily for computer games, which are increasingly being used for non-graphical parallel computing (Owens et al., 2007). The chips contain multiple processor cores (512 in the current state of the art designs) and parallel programming follows the SIMD model (single instruction, multiple data). This model is particularly well adapted to vectorised algorithms. Thus we expect that our algorithms could be ported to these chips with minimal changes, which could yield considerable improvements in simulation speed for considerably less human and financial investment than clusters. We have already implemented the state update part on GPU and applied it to a model fitting problem, with speed improvements of 50 to 80 on the GPU compared to single CPU simulations (Rossant et al., 2010). The next stage, which is more challenging, is to simulate discrete events on GPU.

Another challenge is to incorporate integration methods that use continuous spike timings (i.e., not bound to the time grid), which have been recently proposed by several authors (D'Haene and Schrauwen, 2010; Hanuschkin et al., 2010). The first required addition is that spikes must be communicated along with their precise timing. This could be simply done in a vectorised framework by transmitting a vector of spike times. Dealing with these spike timings in a vectorised way seems more challenging. However, in these methods, some of the more complicated operations occur only when spikes are produced (e.g. spike time interpolation) and thus may not need to be vectorised.

Finally, we only addressed the simulation of networks of single-compartment neuron models. The simulation of a multicompartmental models could be vectorised over compartments, in the same way as the algorithms we presented are vectorised over neurons. The main difference would be to design a vectorised algorithm for simulation of the cable equation on the dendritic tree. As this amounts to solving a linear problem for a particular matrix structure defining the neuron morphology (Hines, 1984), it seems like an achievable goal.

Acknowledgements

The authors would like to thank all those who tested early versions of Brian and made suggestions for improving it. This work was supported by the European Research Council (ERC StG 240132).

A Spike container

The following Python code implements vectorised operations on a circular array (section 2.1.2) and the spike container structure described in section 2.4.

Listing 1: Implementation of circular arrays and spike containers

```python
from scipy import *
from numpy import *

class CircularVector(object):
    def __init__(self, n):
        self.X = zeros(n, dtype=int)
        self.cursor = 0
        self.n = n

    def __getitem__(self, i):
        return self.X[(self.cursor+i)%self.n]

    def __setitem__(self, i, x):
        self.X[(self.cursor+i)%self.n] = x

    def endpoints(self, i, j):
        return (self.cursor+i)%self.n, (self.cursor+j)%self.n

    def __getslice__(self, i, j):
        i0, j0 = self.endpoints(i, j)
        if j0>=i0:
            return self.X[i0:j0]
        else:
            return hstack((self.X[i0:], self.X[:j0]))

    def __setslice__(self, i, j, W):
        i0, j0 = self.endpoints(i, j)
        if j0>i0:
            self.X[i0:j0] = W
        elif j0<i0:
            self.X[i0:] = W[:self.n-i0]
            self.X[:j0] = W[self.n-i0:]

class SpikeContainer(object):
    def __init__(self, n, m):
        self.S = CircularVector(n+1)
        self.ind = CircularVector(m+1)

    def push(self, spikes):
        ns = len(spikes)
        self.S[0:ns] = spikes
        self.S.cursor = (self.S.cursor+ns)%self.S.n
        self.ind.cursor = (self.ind.cursor+1)%self.ind.n
        self.ind[0] = self.S.cursor

    def __getitem__(self, i):
        j = self.ind[-i-1]-self.S.cursor
        k = self.ind[-i]-self.S.cursor+self.S.n
        return self.S[j:k]

    def __getslice__(self, i, j):
        k = self.ind[-j]-self.S.cursor
        l = self.ind[-i]-self.S.cursor+self.S.n
        return self.S[k:l]
```

B Example simulation

The following Python code illustrates many aspects discussed in this article by implementing the simulation of a complete network of spiking neurons. The model corresponds to the CUBA benchmark described in (Brette et al., 2007). It is a network of integrate-and-fire neurons with exponential excitatory and inhibitory currents and sparse random connectivity. It also includes homogeneous synaptic delays and refractoriness. The program simulates simulates the network for 400 ms using vectorised algorithms and displays the raster plot and a sample voltage trace.

Listing 2: Complete example simulation

```
from spikecontainer import *
from random import sample
from scipy import random

N = 4000          # number of neurons
Ne = int(N*0.8)   # excitatory neurons
Ni = N-Ne         # inhibitory neurons
mV = ms =1e-3     # units
dt = 0.1*ms       # timestep
taum = 20*ms      # membrane time constant
taue = 5*ms
taui = 10*ms
p = 80.0/N        # connection probability (80 synapses per neuron)
Vt = -1*mV        # threshold = -50+49
Vr = -11*mV       # reset = -60+49
we = 60*0.27/10   # excitatory weight
wi = -20*4.5/10   # inhibitory weight
delay = 5*ms
refractory = 5*ms
duration = 400*ms

# Spike container
delaysteps = int(delay/dt)
refracsteps = int(refractory/dt)
maxspikes = (N*delaysteps)/refracsteps+1
spikecontainer = SpikeContainer(maxspikes, delaysteps)
```

```
# Update matrix
A = array([[exp(-dt/taum), 0, 0],
           [taue/(taum-taue)*(exp(-dt/taum)-exp(-dt/taue)), exp(-dt/taue), 0],
           [taui/(taum-taui)*(exp(-dt/taum)-exp(-dt/taui)), 0, exp(-dt/taui)]]).T

# State variables
S = zeros((3, N))

# Initialisation
S[0, :] = rand(N)*(Vt-Vr)+Vr # Potential: uniform between reset and threshold

# Connectivity
We_target = []
We_weight = []
for _ in range(Ne):
    k = random.binomial(N, p, 1)[0]
    target = sample(xrange(N),k)
    target.sort()
    We_target.append(target)
    We_weight.append([1.62*mV]*k)
Wi_target = []
Wi_weight = []
for _ in range(Ni):
    k = random.binomial(N, p, 1)[0]
    target = sample(xrange(N),k)
    target.sort()
    Wi_target.append(target)
    Wi_weight.append([-9*mV]*k)

# Simulation
spike_monitor = [] # Empty list of spikes
trace = [] # Will contain v(t) for each t (for neuron 0)
t = 0*ms
while t<duration:
    # STATE UPDATES
    S[:] = dot(A, S)
    # Threshold
    all_spikes, = (S[0, :]>Vt).nonzero()
    spikecontainer.push(all_spikes)
    # PROPAGATION OF SPIKES
    delayed_spikes = spikecontainer[-delaysteps]
    # Excitatory neurons
    spikes = delayed_spikes[delayed_spikes<Ne]
    for i in spikes:
        S[1, We_target[i]] += We_weight[i]
    # Inhibitory neurons
    spikes = delayed_spikes[delayed_spikes>=Ne]-Ne
    for i in spikes:
        S[2, Wi_target[i]] += Wi_weight[i]
    # Reset neurons after spiking
    S[0, spikecontainer[0:refracsteps]] = Vr
    # Spike monitor
    spike_monitor += [(i, t) for i in all_spikes]
    # State monitor
    trace.append(S[0, 0])
    # Update time
    t += dt
```

```
# Plot output
subplot(211)
i, t = zip(*spike_monitor)
plot(array(t)/ms, i, '.k')
subplot(212)
plot(arange(len(trace))*dt/ms, array(trace)/mV, '-k')
show()
```

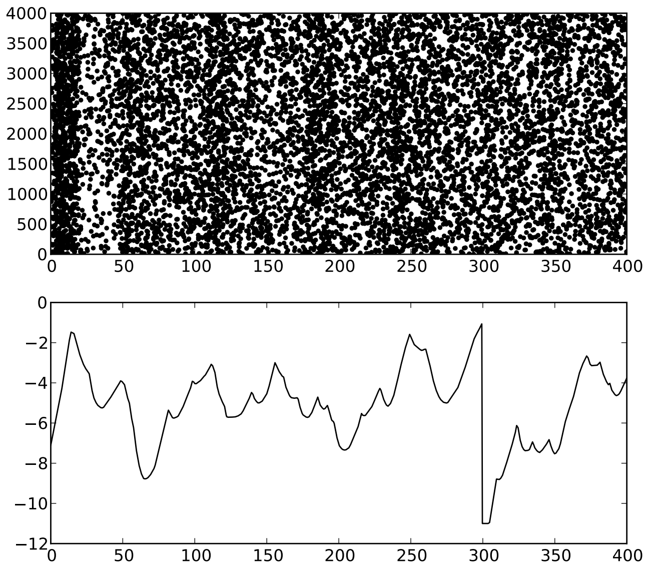

Figure 12: Output of example simulation. Top: Raster plot, showing all spikes produced by the 4000 neurons during the simulation (time on the horizontal axis). Bottom: Sample voltage trace of a neuron during the simulation. Membrane potential is in mV, time is in ms. Note that the threshold is -1 mV, and a spike is produced at time $t = 300$ ms.

References

Bower, J. M. and D. Beeman (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System* (2nd ed.). Springer.

Brette, R., M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris, M. Zirpe, T. Natschläger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. E. Boustani, and A. Destexhe (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience 23*, 349–98.

Cannon, R. C., M. Gewaltig, P. Gleeson, U. S. Bhalla, H. Cornelis, M. L. Hines, F. W. Howell, E. Muller, J. R. Stiles, S. Wils, and E. D. Schutter (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics 5*(2), 127–138. PMID: 17873374.

Carnevale, N. T. and M. L. Hines (2006). *The NEURON Book*. Cambridge University Press.

Davison, A. P., D. Brderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger (2008). PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics 2*, 11.

Deneve, S. (2008). Bayesian spiking neurons I: inference. *Neural Computation 20*(1), 91–117.

Destexhe, A., Z. Mainen, and T. Sejnowski (1994a). An efficient method for computing synaptic conductances based on a kinetic model of receptor binding. *Neural Computation 6*(1), 14–18.

Destexhe, A., Z. F. Mainen, and T. J. Sejnowski (1994b, August). Synthesis of models for excitable membranes, synaptic transmission and neuromodulation using a common kinetic formalism. *Journal of Computational Neuroscience 1*(3), 195–230.

D'Haene, M. and B. Schrauwen (2010, June). Fast and exact simulation methods applied on a broad range of neuron models. *Neural Computation 22*(6), 1468–1472.

Djurfeldt, M., M. Djurfeldt, and A. Lansner (2007). Workshop report: 1st INCF workshop on large-scale modeling of the nervous system. *Nature Precedings*.

Eppler, J. M., M. Helias, E. Muller, M. Diesmann, and M. Gewaltig (2008). PyNEST: a convenient interface to the NEST simulator. *Frontiers in Neuroinformatics 2*, 12.

Garny, A., D. P. Nickerson, J. Cooper, R. W. dos Santos, A. K. Miller, S. McKeever, P. M. F. Nielsen, and P. J. Hunter (2008, September). CellML and associated tools and techniques. *Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences 366*(1878), 3017–3043. PMID: 18579471.

Gewaltig, O. and M. Diesmann (2007). NEST (NEural Simulation Tool). *Scholarpedia 2*(4), 1430.

Giugliano, M. (2000, April). Synthesis of generalized algorithms for the fast computation of synaptic conductances with markov kinetic models in large network simulations. *Neural Computation 12*(4), 903–931.

Giugliano, M., M. Bove, and M. Grattarola (1999, August). Fast calculation of short-term depressing synaptic conductances. *Neural Computation 11*(6), 1413–1426.

Goddard, N. H., M. Hucka, F. Howell, H. Cornelis, K. Shankar, and D. Beeman (2001). Towards NeuroML: model description methods for collaborative modelling in neuroscience. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences 356*, 1209–28.

Goodman, D. and R. Brette (2008). Brian: a simulator for spiking neural networks in python. *Frontiers in Neuroinformatics 2*, 5.

Goodman, D. F. M. and R. Brette (2009, September). The Brian simulator. *Frontiers in Neuroscience 3*(2), 192–197.

Hanuschkin, A., S. Kunkel, M. Helias, A. Morrison, and M. Diesmann (2010). A general and efficient method for incorporating precise spike times in globally time-driven simulations. *Frontiers in Neuroinformatics 4*(0), 12.

Hines, M. (1984, February). Efficient computation of branched nerve equations. *International Journal of Bio-Medical Computing 15*(1), 69–76.

Hines, M. L. and N. T. Carnevale (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation 12*(5), 995–1007.

Hines, M. L., A. P. Davison, and E. Muller (2009). NEURON and Python. *Frontiers in Neuroinformatics 3*, 1.

Hirsch, M. and S. Smale (1974). *Differential equations, dynamical systems, and linear algebra.* Academic Press.

Izhikevich, E. M. (2006, February). Polychronization: computation with spikes. *Neural Computation 18*(2), 245–282. PMID: 16378515.

Jahnke, A., U. Roth, and T. Schnauer (1999). Digital simulation of spiking neural networks. In *Pulsed neural networks*, pp. 237–257. MIT Press.

Köhn, J. and F. Wörgötter (1998, October). Employing the Z-Transform to optimize the calculation of the synaptic conductance of NMDA and other synaptic channels in network simulations. *Neural Computation 10*(7), 1639–1651.

Loebel, A. and M. Tsodyks (2002). Computation by ensemble synchronization in recurrent networks with synaptic depression. *Journal of Computational Neuroscience 13*(2), 111–124.

Lytton, W. W. (1996). Optimizing synaptic conductance calculation for network simulations. *Neural Comput 8*(3), 501–9.

Markram, H., Y. Wang, and M. Tsodyks (1998, April). Differential signaling via the same axon of neocortical pyramidal neurons. *Proceedings of the National Academy of Sciences of the United States of America 95*(9), 5323–5328.

Mongillo, G., O. Barak, and M. Tsodyks (2008, March). Synaptic theory of working memory. *Science 319*(5869), 1543–1546.

Morrison, A., A. Aertsen, and M. Diesmann (2007, June). Spike-Timing-Dependent plasticity in balanced random networks. *Neural Computation 19*(6), 1437–1467.

Morrison, A., M. Diesmann, and W. Gerstner (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics 98*(6), 459–478.

Morrison, A., C. Mehring, T. Geisel, A. Aertsen, and M. Diesmann (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput in press*. undefined Advancing the boundaries of high connectivity network simulation with distributed computing.

Morrison, A., S. Straube, H. E. Plesser, and M. Diesmann (2007). Exact subthreshold integration with continuous spike times in discrete-time neural network simulations. *Neural Computation 19*, 47–79.

Morse, T. (2007). Model sharing in computational neuroscience. *Scholarpedia 2*(4), 3036.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell (2007). A survey of General-Purpose computation on graphics hardware. *Computer Graphics Forum 26*(1), 80–113.

Platkiewicz, J. and R. Brette (2010). A Threshold Equation for Action Potential Initiation. *PLoS Comput Biol 6*(7), e1000850.

Plesser, H. E. and M. Diesmann (2009, February). Simplicity and efficiency of Integrate-and-Fire neuron models. *Neural Computation 21*(2), 353–359.

Protopapas, A., M. Vanier, and J. Bower (1998). *Methods in Neuronal Modeling: from Ions to Networks*, Chapter Simulating large networks of neurons, pp. 461.

Rossant, C., D. F. M. Goodman, J. Platkiewicz, and R. Brette (2010). Automatic fitting of spiking neuron models to electrophysiological recordings. *Frontiers in Neuroinformatics 4*, 2.

Rotter, S. and M. Diesmann (1999, November). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biological Cybernetics 81*(5-6), 381–402. PMID: 10592015.

Sanchez-Montanez, M. A. (2001). Strategies for the optimization of large scale networks of integrate and fire neurons. Volume 2084/2001 of *Lecture Notes in Computer Science*. Springer-Verlag. undefined IWANN Strategies for the Optimization of Large Scale Networks of Integrate and Fire Neurons 0302-9743.

Schutter, E. D. (2008, May). Why are computational neuroscience and systems biology so separate? *PLoS Comput Biol 4*(5), e1000078.

Song, S., K. D. Miller, and L. F. Abbott (2000). Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neurosci 3*, 919–26.

Tsodyks, M., K. Pawelzik, and H. Markram (1998). Neural networks with dynamic synapses. *Neural Computation 10*(4), 821–835.

Tsodyks, M. V. and H. Markram (1997). The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. *PNAS 94*(2), 719–23.