CEMRACS 2017: Network of interacting neurons with random synaptic weights

October 12, 2017

Abstract

Derived from the work of Hodgkin and Huxley, several examples of the fokker planck equations can be used to model the evolution in time of the potential of the membrane of a neuron The solutions to the Fokker-Planck equation describing the behaviour of the potential of the membrane of a neuron can sometimes degenrate and cease to exist after a finite time. The conditions for this phenomenon called blow up to exist have been studied analytically in several papers. During the 2017 session of the cemracs we have implemented an algorithm for simulating a network of neurons of potentials driven by this equation. One common way to model neurons interaction in the cortex is to use a mean-field equation describing the behaviour of neurons. This kind of models have been proposed several times since the analogy between neurons and a simple electric circuitry by Hodgkin and Huxley and generalised to an equation describing the evolution of the membrane's potential of each neuron in an infinite network. In this report we are interested in describing an algorithm, based on a thinning method, that will help simulate such a network for a big number of neurons. After a description of the model, we explain in details the algorithm developed and implemented, and the choices (theoretical and practical) we made. We finish by a setting up a benchmark for experimenting with the model and showing some results obtained with the simulation.

Contents

0.1 Introduction

On of the first models for neurons was introduced by Louis Lapicque in 1907 and called Integrate and Fire. Neurons are represented in time by the simple electrical equation

$$I(t) = C_m \frac{dV_m}{dt*} \tag{1}$$

which is just the time derivati*ve of the law of capacitance. A positive current being applied to the membrane, it's potential is going to increase until it reaches a certain threshold value V_T, at which point a dirac delta function happens and the voltage of the mambrane is reset to a resting potential.
From this model is the basis of a large variety of other neural models invented to model more precisely certain behaviours of neurons and neural networks, as memory, leaking, etc.

0.2 Model and background

In this study we are interrested in a large (possibly infinite) network of interacting integrate and fire neurons. [**?**] and [**?**] proposed an equation describing

the evolution in time of the potential V_i of the i^{th} neuron in a network of N

$$\begin{cases} \frac{d}{dt}V_i(t) = -\lambda V_i(t) + \frac{\alpha}{N}\sum_j \sum_i \delta_0(t-\tau_k^j) + \frac{\beta}{N}\sum_{j\neq i} V_j(t) + I_i^{ext}(t) + \sigma\mu_i(t) & \text{if } V_i < V_T \\ V_i(t^+) = V_R & \text{if } \mathrm{V}_i \text{ reaches } \mathrm{V}_T \text{ at time t} \end{cases} \tag{2}$$

[**?**] and [**?**] gives more detail about the physical signification of the terms in the equation. As an overview, $V_i(t)$ is the function associated with the evolution in time of the potential of the membrane of the neuron i, $I_i^{ext}(t) + \sigma\mu_i(t)$ is the effect of electrical currents outside of the studied network (mean value + gaussian white noise), $\frac{\alpha}{N}\sum_j \sum_i \delta_0(t-\tau_k^j)$

0.2.1 Motivations

Here we are interested only on the spiking behaviour of the system, and so the mean field equation considered thereafter in this document is a simplified form of the one presented just above, as some terms are irreleveant to the question. In this paper we are going to focus on what sets of parameters favors the apparition of a blowing up of the system. The influence of the interaction term is a big part of the study, but the influence of other parameters, such as the b function, the noise, and the topology of the network are also looked at.
In the deterministic case, the blow up is quite easily defined by the limit to a time value t_b of the variations in the spike rate equals to infinity, in other words

$$\lim_{t\to t_b} \frac{de}{dt} = \infty \tag{3}$$

This behaviour of the PDE has been described in [**?**], while others have been interrested in this exact phenomenon from a PDE perspective. Indeed, systems ruled by this kind of equations do not automatically blow-up; actually some conditions on the parameters must be met in order to observe this phenomenon.

0.2.2 Mean field equation

That being said, we can now pose the real equation under study in this report.

$$V_t^i = V_0^i + \int_0^t b(V_s^i)ds + \sigma W_t^i + \sum_{j=1}^{N} J^{j\to i} M_t^j - M_t^i(V_T^i - V_R^i)$$

Here b is Lipschitz continuous, and $\forall V, b(V) = -\lambda(V-a), (\lambda, a) \in \mathbb{R}_+$. This function will make the potential drift towards the value a. The J term models the interactions between neurons, their values depends on the number of neurons in the system and which neurons are actually involved in the interaction. Throughout the event several values of interaction have been tested. They generally are Bernoullis variables, determined at the creation of the system.

0.3 The algorithm

Before showing any result a discussion is needed on the algorithmic and the implementation part. The goal is to be able to simulate a system driven by the equation described above in reasonnable times and with as many neurons as possible: the mean field hypothesis makes more sens in bigger networks and biological networks are composed of a tremendous amount of neurons anyway (100000 for drosophilia, 86 billions for human beings).
For this, even though the algorithms presented here are language-agnostic, the implementation has been made in C (actually first in Java and then translated in C). The code is available at `https://github.com/mascartcyrille/cemracs`.

The first discussion is on the threshold. There are indeed two ways of modelling it, as a "hard" threshold or as a "soft" one. Hard here means that the potential of a neuron is reset at the exact moment it is detected to be higher than the threshold. This means that if several neurons' potential cross the threshold at the same time, they all are going to fire alltogether. In the case of a blowing-up of the system, this can lead to a "cascade", a case where a group of neurons are firing indefinitely at a given time. The system then has no solution after that time.
The soft threshold is a more generic case where the neurons do not automatically fire when their potential increases at or above the threshold, but instead acquire a chance to spike. The chance of spiking follows a given distribution that actually increases polynomially the chance of firing with the amount of potential above the threshold.
The stochastic model was chosen as the object of study, being more generic than it's deterministic counterpart. However this necessitates to explicit some parameters and to refine some definitions that do not fit this model.

0.3.1 Definitions

With this approach some definitions given before are no longer appliable to the system.
The threshold is then defined as:
At any given time, any neuron i of potential V_i has a chance of firing which is function of

$$f(V_i) = S * (Pos(V_i - V_T))^E, \text{ where } Pos(x) = \begin{cases} x & \text{if } x > 0 \\ V_R & else \end{cases}$$

V_T is chosen equal to one, and V_R is equal to zero for all neurons (for simplicity purposes), but they could be chosen more randomly. A threshold value too low (too close to the reset value) is problematic, as it means that potentials are going to reach their threshold quickly after a neuron has spiked (in which case the system is "stuck" in a perpetual series of spikes). By design only one neuron

can fire at a given time (the spike trains are point processes), and if a cascade is cannot normally happen (except in cases where some values are rounded to zero due to the limitations in floating point precision, but more on that later) that does not solve the issue of blow-up, as a similar phenomenon can still occur (the rate will not tend to infinity, still it tends to very high values). The blow-up is indeed newly defined as (Pertinence de comparer à N lorsque tous les neurones ne spikent pas ? Pourquoi pas plutôt quelque chose du type $1/\delta_N -> \infty$, avec δ_N l'intervalle entre deux spikes ?):

$$\frac{1}{\delta_N} > Constant, \text{ typically N}$$

where δ_N is an interval between two spikes (or a mean interval, or a moving average). The constants S and E are chosen so that the chances of fire are high even for values of the potential low above the threshold (typically, $S = 10^5$ and $E = 5$, while values of potentials at spiking times are around [CALCULER VALEUR PRECISE, DE MEMOIRE 1.4]).

The system is driven by a stochastic PDE and discontinuities are randomly introduced depending on the value of parts of the system. There are two common ways of simulating a stochastic PDE. The Euler-Maruyama method for numerical approximation of stochastic differential equations results in the construction of a Markov chain on the interval $[0, T]$ of simulation. While quite simple to set up and understand, it is wastefull for processes that spend a lot of time not changing, in this case not spiking. In addition, the resolution of the time interval shall be low enough to capture the meaningful variations in the system, here the moments of spikes and the blowing-up behaviour, as well as to keep the simulation precise enough. Spikes are diracs, discontinuities of the system, and the blow up happens when the firing rate tends to infinity, or to spell it otherwise, when the time interval between spikes tends to zero.
The second method is the thinning, which is a common method for generating random numbers from distribution of not well defined cumulative distribution function `http://www.aip.de/groups/soe/local/numres/bookcpdf/c7-3.pdf`. Here the stochastic process is the network of neurons, and its intensity directly depends on the potentials of the neurons, following the function $f : \mathbb{R} \to \mathbb{R}^+$ described above. In this method two numbers (for the one dimension case) are generated, $(t, u) \in \mathbb{R} + \times [0, 1]$. They represent a point in the plane and depending on whether the point can be placed under the curve of the desired cumulative distribution function or not the point will be declared accepted or rejected. In order to achieve this in practice the value of $\frac{f(x)}{f_{max}(x)}$ is compared to a number uniformly generated between zero and one. The approximation function is used in order to increase the probability of generating a point under the curve (on the full spectrum of real positive numbers, the chances are null).

[//TODO: INCLUDE A GRAPHIC EXAMPLE OF THE PROCEDURE, ALIKE THE ONE DRAWN ON BOARD]

k	**chance** ($\approx$)
1	3.173105e-01
2	4.550026e-02
3	2.699796e-03
4	6.334248e-05
5	5.733031e-07
6	1.973175e-09

Table 1: Chances of generating a number of absolute value k times above the variance, knowing that $P(X < \mu - k\sigma ORX > \mu + k\sigma) = 2 - 2\phi(k)$ (computed with R using *pnorm* for a centered normalised law)

The f_{max} here can be computed by a simplified, limit in time version of the model. Indeed, a rough estimate can be created by dividing the equations in sums of which the maximum value in time will be taken.

$$f_{max} = \max_t(a + \exp^{-\lambda t}(y_t^i - y_S^i)) + \max_t(\sigma\mathbb{N}(0, \frac{1 - \exp^{-2\lambda t}}{2\lambda}))$$

The maximum proposed here is not correct. Indeed, there are no boundaries for a normally distributed random number, hence the white noise cannot be bounded and the variations for the potentials are in $\mathbb{R} \cup \{-\infty, \infty\}$... But! The chance of generating a number of absolute value larger than the variance decreases the farther away this number is from the variance. Thus it is possible to consider, with a certain degree of confidence, that no numbers will be generated outside of a predefined interval, the bounds of which will constitute our maximum. More precisely, given a random variable X following a normal law $X\ \mathbb{N}(\mu, \sigma^2)$, then it is possible to compute $P(\mu + k\sigma \leq X)$. The table 1 gives values of the chances of generating a number outside the interval $[\mu - k\sigma, \mu + k\sigma]$.
In the current implementation k has been fixed at 5, as the number of neurons will hardly be higher than 10^5 for memory reasons (the graph of interaction takes $p \times N^2 \times 64$ bits of ram memory, p being the probability of connection between two neurons in a Erdos-Renyi graph, 64 being the number of bits necessary for representing an integer on a typical computer nowadays and for $N = 10^5$ this value is higher than the typical amount of ram available on a computer, even for dedicated comptuting machines). There is also an influence of the number of spikes (accepted or rejected) generated during the simulation: each time the simulator decides whether the spike is accepted or not the true value of the potential is computed therefore the value of the noise is computed and there is a chance of generating a normally distributed number higher than the bound we have set.

So tosummarize, we have chosen to use a thinning procedure to compute the solution of the stochastic partial differential equation described in the **??**. The thinning procedure necessitates the use of an approximation function, which must be in any point higher than the function to simulate/compute. Even

though a part of our model is a gaussian white noise, which is unbounded, it is still possible to use a pseudo-boundary for the noise, that is to say a number that will constitute an upper-limit with a certain degree of confidence. The limit directly depends on the confidence interval we want for our simulation, so this solution is both easy and flexible. Computing this pseudo-boundary necessitates only values that are necessary to compute the value of the gaussian white noise, so this solution does not increase the amount of computations needed. Actually it is much less expensive, in processor time, to compute the value of the upper-limit, as the actual value of the variance (which relies on an exponential in our case) is not needed (a maximum in time of the variance is enough, and adds in the confidence). Also, it is important to remark that even if we have focused on giving an upper-bound to the noise, there are other parts in our model that are approximated too, and so values of the noise not too much above the upper-limit may not make our approximation function f_{max} lesser than the actual potential, so the solution is robust even to small errors.

Finally, we can remark that we are sampling the noise anyway, meaning there is no way for us to know that even if the noise at the time of spike is indeed in the boundaries, it has not crossed the boundaries during the time inbetween two consecutive spikes. So what if there are errors in the evaluation of the noise? Two cases emerge. First, there is a bad approximation of the noise of the spiking neuron. So this neuron is certainly the good one to pick for a spike but it may have spiked a bit earlier if we had the information of the real value of its potential. The second case is more criticial, as this is the case where a neuron has been chosen for spiking, but the potential of one of the non chosen neurons is higher than expected, meaning it had actually more chance to spike. Of course the neuron we have chosen may actually still be the good one, then the error does not matter at all, the next computation of the potential shall rectify the mistake. Yet, even if we imagine that having another estimation for the potential would have changed the spiking neuron, given that we are dealing we at minimum 10,000 neurons, and more probably 100,000 neurons, and generating around the same number of spikes, having even a handfull of mistakes would not mean a large deviation of the final result (if a neuron should have spiked but does not, and this because of the noise, not the configuration of the network or the intensity of the spikes or whatever, then it will spike just a bit later, and excepting in the case where we speak about very critical neurons of very specific networks, this kind of mistakes can be considered as a good tradeof for speed).

At this stage we have a first draft for the thinning procedure algorithm, but we can improve it a lot. For now we have focused on improving the efficiency of and discussing about the quality of the approximation function f_{max}, but f is not the real equation simulated here, it just helps defining a probability for a neuron to spike, depending on the value of its potential. But as the potential evolves in time, the approximation is always false, and by a bigger margin at the beginning of time than at infinity, since the approximation function is built

```
Initialize all parameters
Label: do {
  do {
    determine an interval [t{n-1},t_n] on which sampling
    compute the array of f_max(Y_{t_{n-1}}) and
∑ f_max(Y_{t_{n-1}}) on interval $[t_{n-1},t_n]$
    t_n ~ t_{n-1} + 𝒪(∑ f_max)
  while( not in good interval or all f_max are at 0 )
  u ~ 𝕌([0,1]) → ℙ(spiking neuron is neuron i) = f^i_max(Y^i_{t_n}) / ∑_j^N f^j_max(Y^j_{t_n})
  u ~ 𝕌([0,1]) → ℙ(accepting spike of neuron i) = f^i(Y^i_{t_n}) / f^i_max(Y^i_{t_n})
  if( spike is accepted ) {
    update the potentials of all postsynaptic neurons
  }
} while( do not have the good amount of accepted spikes )
```

Figure 1: Pseudo code of the thinning algorithm used for simulating the system

using the values for $t \to \infty$. Unfortunately, the values of t (the x-axis part of the points generated for the thinning method) have a lot more chance to be small as they are generated using an exponential distribution (we are dealing with poisson processes so the spikes must be exponentially distributed in time [enfin ici du moins en l'occurrence c'est en temps]). This is also an issue for the random generation, as pseudo-random number generators cycle and generating more unuseful numbers may lead the generator to cycle.

A common solution to avoid this is to bound the research of accepted points in time: we define a time interval of research and then look for points in this interval. Once a certain condition is met (for instance here every time we have an accepted point or when the time for the next potential spike is greater than the time limit) we stop or change the time interval. This way the approximation function is way closer to the real function and we can drastically improve the amount of accepted points.

[//TODO: INSERT HERE GRAH OF THINNING METHOD WITH TIME INTERVALS]

0.3.2 Algorithm in pseudo code

Finally here is, in pseudo code, the algorithm used for simulating the equation driving our model.

0.3.3 Improving efficiency

[//TODO: PENSER A METTRE UN PEACH SUR OPENMP, LES ALTERNATIVES ET PK ON (JE) L'A CHOISIS] This subsection will mainly focus on speed and memory usage. The algorithm presented above works perfectly except that the computations involved here can be very long in processor time, and that the amount of memory for running it is potentially tremendous. On the speed part, the algorithm has been implemented in C and optimised using -O3 in order to take advantage of the gain of speed of the low level languages. Dedicated libraries (here openmp) can and have been used in order to improve the speed of some parts of the simulation, in the parts where a state variable needs to be changed for all or a large amount of neurons. Some specific interaction cases, like full connection (including selfconnections), complete interaction graph and independence (no connections) are encoded so that the connection graph is not needed, improving speed and memory consumption. but on the generic case, even though the full matrix of interactions is not stored (only the relevant parts), the memory for storing the interaction graph takes around $p * N^2$, where p is the probability of interaction and N the number of neurons. For even 10^5 neurons that makes about 1 GB, given 1 byte is enough to store the index of neurons (and the addresses in memory)(spoiler: it is not, and takes rather 4 times more) and with a probability of connction of 0.1... where the probability of connection was 1 in the mathematical papers.

Still, when the interaction graph is too big but there is no way to skip the generation of the interaction graph, there is a way to save a lot of memory by storing seeds instead of a boolean about whether there is an interaction or not. When generating an interaction graph with a random connection probability, for each neuron, instead of storing an array containing alll the postsynaptic neurons, one can rather store the state of the random number generator (for instance, in Mersenne Twister, it is about 128 bits, and it is one of the largest state for a classical random number generator) just before randomly chosing the postsynaptic neurons. This way, each time a neuron is spiking, one can regenerate the same graph, and more interestingly the specific part of the interaction graph of interest (that is to say only the postsynaptic neurons of the spiking neurons, and no the entire graph of interaction). While a lot slower, this method is probably one of the most efficient in memory, taking advantage of the pseudo character of the random number generation in computer science.

This method is also parallizable with openmp or any library alike it, but it is more difficult to achieve if one want to guarrantee the reproducibility of the results. The parallelization is possible because if one cuts the array of the postsynaptic interactions in k parts, then k random number generators can be used for generating the interaction graph. The issue here is on the initialization of the random number generators, as we want to avoid the case where there is an overlapping in the sequences in use by two threads at the same time. A RNG like the Mersenne Twister, with a very large pseudo period is great for this

purpose, as it allows to cut the pseudo period so that two processes correctly seeded will never overlap. This considerations are also needed if one wants to run several simulations in parallel. [METTRE PUBLI BENNIE]

0.3.4 Implementation

As said before, the algorithm was finally implemented in C (and prototyped in Java). There are several reasons for that. The C language is close enough to the machine language to be one of the fastest possible and allows a precise management of the memory usage. Most compiler (as gcc, the one we used) also have options for improving the speed and memory consumption of the program. These options are also supported by the version of the Mersenne-Twister random number generator used in the implementation. There are two of them, on for integer generation and one for double precision generation, and they can be found at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html`. O this site there are more information about Mersenne-Twister generators in general, as this page has been created and is maintained by one of their creator, Makoto Matsumoto.

The computations at stake here involve potentially both very low and very high values. For instance the time of spikes are exponentially distributed. They are obtained using an exponential distribution of parameter the sum of all the approximation functions (on the current interval of work). This value can typically raise up to 10^5 or 10^6 (consider the same number of neurons, their potential close to the threshold).

For this reason some

0.3.5 A word about the rng

Mersenne twister, and not xorshift because the mersenne allows for a reproduction of the simulations, which is also necessary for a part of the algorithm. It is well known, and has a huge period which is important for a part of the algorithm. Also there are a lot of good implementations, some of the creator itself who actively update them when bugs or improvements are found. The method also allow for good generation of random real numbers (not only integers)

A related subject is the distribution followed by the random numbers. The algorithm presented earlier needs several random numbers from various distributions, none of them being too exotic (uniform numbers with arbitrary upper-bound and normal numbers centered on zero with arbitrary variance), but as the library used only provide uniform random series of 32 or 64 bits and random floating point numbers in [1,2) or [0,1], [0,1), (0,1], (0,1), we must implement a way to generate the numbers drawn from our arbitrary distributions. In both cases a sampling method (again) is used, and we are going to describe them.

First the uniform case. We have a random integer generator, and we assume it gives perfectly random numbers in $0, 1, ..., M$, while we want uniform

numbers in $0, 1, ..., U$. A common method to achieve this is to use a modulo: $\mathbb{U}([0, M])mod[U + 1]$. The issue here is that unless M is a multiple of $U + 1$, the uniformity of the numbers is lost using this method. The case in which the method works can be used as a trick for guarrantying uniformity though. Indeed, if M is not a multiple of $U+1$, there must exist one that is lower than M but close anyway. Knowing that, we can simply take any number that is lower than $U + 1$ and in the rare cases a number greater than $U + 1$ is drawn, we just have to reject it and draw another one. Of course the maximum value M must be quite high, compared to $U + 1$ in order for this method to be effective. For instance, if $U + 1 =$, there is a 49% of rejection. If the rest of the euclidean division of M bu $U + 1$ is U, the

0.4 Experiments