Implementaion of the Potjans Diesmann microcircuit model using the classical Galves-Löcherbach neuron model

Nilton Liuji Kamiji, Christophe Pouzat, Aline Duarte, Antonio Roque, Antonio Galves

July 27, 2022

Contents

List of Figures

1 Model considered

1.1 The original PD model

The PD model (Potjans and Diesmann, 2014) consists of 8 neuronal populations, representing excitatory and inhibitory populations of 4 cortical layers, namely, layers 2/3 (L23e/i), 4 (L4e/i), 5 (L5e/i) and 6 (L6e/i), where e denotes excitatory and i inhibitory. The number of neurons of each population is shown in table 1, which totalizes 77169 neurons.

Table 1: Number of neurons per cortical layer

L23e	L23i	L4e	L4i	L5e	L5i	L6e	L6i
20683	5834	21915	5479	4850	1065	14395	2948

1.1.1 Membrane potential dynamics

All neurons are described with the same LIF (leaky integrate-and-fire) dynamics (eq. (1))

$$\begin{aligned}
\frac{dV_m(t)}{dt} &= \begin{cases} 0 & \text{if neuron is refractory} \\ -\frac{V_m(t)}{\tau_m} + \frac{S^i_{syn}(t)}{C_m} & \text{otherwise} \end{cases} \\
\frac{dI_{syn}(t)}{dt} &= \begin{cases} 0 & \text{if neuron is refractory} \\ -\frac{I_{syn}(t)}{\tau_{syn}} + \sum_j W_j & \text{otherwise} \end{cases} \\
\text{if } V^i_m(t) \geq V_{th} & \quad \text{neuron spiked and refractory}
\end{aligned} \tag{1}$$

where $V_m(t)$ is the membrane potential, $\tau_m = 10$ ms is the membrane time constant representing the leakage effect, $C_m = 250$ pF the membrane capacitance, $V_{th} = 15$ mV the threshold potential in which a neuron fires when its membrane potential exceeds this value, $I_{syn}(t)$ the membrane current elicited by synaptic inputs from pre-synaptic neurons j (W_j), and $\tau_{syn} = 0.5$ ms the synaptic time constant indicating the dynamics of synaptic receptor channel. The refractory period (τ_{ref}), the period in which a neuron stays silent after emmiting a spike, was 2 ms.

1.1.2 The graph of interaction

Neurons are connected following the connection probability between neuronal populations shown in table 2. The resulting graph of interaction is a directed graph with nearly 300 million edges.

Moreover, synaptic connections between neurons j and i are described by the synaptic weights ($w_{j\to i}$) and transmission delay ($d_{j\to i}$), drawn from a clipped normal distribution, and are different for excitatory and inhibitory synapses as shown in table 3. dt is the simulation time step, indicating that synapse transmission requires at least one time step.

Remark 1 A synaptic weight of 87.8 pA will elicit a maximum membrane potential change of approximately 0.15 mV.

Table 2: Connection probability between neuronal populations

		from							
		L23e	L23i	L4e	L4i	L5e	L5i	L6e	L6i
	L23e	0.101	0.169	0.044	0.082	0.032	0.0	0.008	0.0
	L23i	0.135	0.137	0.032	0.052	0.075	0.0	0.004	0.0
	L4e	0.008	0.006	0.050	0.135	0.007	0.0003	0.045	0.0
to	L4i	0.069	0.003	0.079	0.160	0.003	0.0	0.106	0.0
	L5e	0.100	0.062	0.051	0.006	0.083	0.373	0.020	0.0
	L5i	0.055	0.027	0.026	0.002	0.060	0.316	0.009	0.0
	L6e	0.016	0.007	0.021	0.017	0.057	0.020	0.040	0.225
	L6i	0.036	0.001	0.003	0.001	0.028	0.008	0.066	0.144

Table 3: Synaptic weight and delay. Synaptic weights are clipped at 0, and synaptic delays are clipped at simulation step ($d_t = 0.1$ ms)

	w_e	$\mathcal{N}(\mu = 87.8, \sigma = 8.8)$ pA; $w_e > 0$
excitatory	d_e	$\mathcal{N}(\mu = 1.5, \sigma = 0.75)$ ms; $d_e \geq dt$
	w_i	$\mathcal{N}(\mu = -351.2, \sigma = 35.1)$ pA; $w_i < 0$
inhibitory	d_i	$\mathcal{N}(\mu = 0.8, \sigma = 0.4)$ ms; $d_i \geq dt$

Remark 2 The number of synapses (K) between any two populations are calculated from:

$$K = \frac{\log(1 - C_a)}{\log(1 - 1/(N_{pre} N_{post}))}, \tag{2}$$

where C_a is the connection probability taken from table 2, N_{pre} and N_{post} the number of neurons in the pre- and post-synaptic population, respectively, taken from table 1. The K pairs are then randomly chosen, so that multiple synapses with different parameter values for any pair can occur. This phenomenon is know as multapses.

Remark 3 The connection weight between neurons of L4e to L23e is doubled, that is, taken from a clipped normal distribution of $\mathcal{N}(\mu = 175.6, \sigma = 17.6)$ pA

1.1.3 Background input

The network is driven by a constant external input that can be described as Poissonian spike trains ($rate$ = 8 Hz) or constant current. The number of external inputs is layer dependent as shown in table 4.

Table 4: Number of external inputs onto each cortical layer

L23e	L23i	L4e	L4i	L5e	L5i	L6e	L6i
1600	1500	2100	1900	2000	1900	2900	2100

Remark 1 In the case of Poissonian spike trains input, every neuron on each layer receives

independent Poissonian spike trains with $rate = 8 \times n$, where n is the number of external inputs, which is the same as generating n Poissonian spike trains of 8 Hz and applying all of it to the neuron. Note that all input has fixed weight and delay values of 87.8 pA and 1.5 ms, respectively.

1.2 The GL model

We consider a network made of (N) neurons in a discrete time setting. We write $I = \{1, \dots, N\}$ the neurons' index set. The neurons can be viewed as the nodes of a *directed graph*. The synapses between neurons / edges of the graph are specified by a quadruple:

$j \in I$ The *.presynaptic neuron*.

$i \in I$ The *postsynaptic neuron*.

$w_{j \to i} \in \mathbb{R}$ The *synaptic weight*, if $w_{j \to i} = 0$ there is no synapse from j to i, if $w_{j \to i} < 0$ the synapse is *inhibitory*, if $w_{j \to i} > 0$ it is *exitatory*.

$d_{j \to i} \in \mathbb{R}^+$ The *synaptic delay*, this accounts for the *propagation time* of the spike from the soma of neuron j to the the synaptic terminal on neuron i, as well as the time for the presynaptic vesicles to fuse upon spike arrival, the transmitter to diffuse and bind to the postsynaptic receptors and the latter to open.

1.3 Spike trains

A *spike train* is associated with each neuron, i, of the network. This is a sequence of random variables $X_t(i)$, where t is the discrete time index, taking value in $\{0, 1\}$. $X_t(i) = 0$ means that neuron i does not spike at time t, while $X_t(i) = 1$ means neuron i spikes at time t.

Last spike time We are going to consider models with membrane potential reset (see next section) following a spike. This will force us to refer to the *last spike time of neuron i before time t, $L_t(i)$*:

$$L_t(i) = \max\{s \leq t : X_s(i) = 1\} . \tag{3}$$

1.4 Membrane potential and rate function

With every neuron i we associate a *Membrane Potential Process* $V_t(i)$ and a *rate function* $\phi_i : \mathrm{R} \mapsto [0, 1]$. Given the $\{V_t(i)\}_{i \in I}$, each neuron spikes at time $t + 1$ **independently of the others** with probability $\phi_i\left(V_t(i)\right)$.

In the absence of input the membrane potential of each neuron decays spontaneously and *deterministically* according to:

$$V_{t+1}(i) = \rho V_t(i), \quad 0 < \rho \leq 1 , \tag{4}$$

this dynamics reflects the membrane potential leakage. ρ could clearly be made neuron specific or specific to a neuron class (exitatory / inhibitory neurons for instance).

1.5 Synaptic input

If $w_{j \to i} \neq 0$, a spike in neuron j will influence the membrane potential of neuron i (and its subsequent probability of spiking) with a delay $d_{j \to i}$. Since we are considering a model with "comprehensive reset following a spike" only the spikes *arriving* after the last spike of neuron i, ($L_t(i)$) are going to contribute. We must therefore only consider a subset of the time indices, namely:

$$\{s : X_s(j) + d_{j \to i} > L_t(i)\} \, .$$

A last feature of synapses that we might want to consider is the fact that synaptic conductances remain open (or active) for a finite amount of time and decay (roughly) mono-exponentially. We will therefore model the effect of a spike in neuron j at time $X_s(j)$ on the membrane potential of neuron i by:

$$w_{j \to i} \xi^{t-s-d_{j \to i}} X_s(j), \; X_s(j) + d_{j \to i} > L_t(i) \text{ and } 0 \leq \xi \leq 1 \, , \tag{5}$$

where ξ accounts for the synaptic conductance dynamics. If it is null since $0^0 = 1$ the synaptic input is limited to the time at which the presynaptic spike arrives. Again parameter ξ could be made synapse or synapse type (exitatory *vs* inhibitory) specific.

The last assumption we need is the **linear summation** on the membrane potential of all the synaptic inputs received by a given neuron in the absence of spike of the latter.

1.6 Overall dynamics

In practice a sequence of Bernoulli random variable is used at each time step: $\{U_t(i)\}_{i \in I}$, $U_t(i) \overset{IID}{\sim} \mathcal{B}$ and the $V_{t+1}(i)$ are given by:

$$V_{t+1}(i) = \begin{cases} 0 & \text{if } U_t(i) \leq \phi_i\left(V_t(i)\right) \\ \rho V_t(i) + \sum_{j \in I} w_{j \to i} \left(\sum_{s=L_t(i)+1-d_{j \to i}}^{t-d_{j \to i}} \xi^{t-s-d_{j \to i}} X_s(j) \right) & \text{otherwise} \end{cases} \tag{6}$$

and

$$X_{t+1}(i) = \begin{cases} 1 & \text{if } U_t(i) \leq \phi_i\left(V_t(i)\right) \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Remark 1 If $\xi = 0$ and $d_{j \to i} = 0$ for all $(i,j) \in I^2$, then Eq. 6 becomes much simpler, namely:

$$V_{t+1}(i) = \begin{cases} 0 & \text{if } U_t(i) \leq \phi_i\left(V_t(i)\right) \\ \rho V_t(i) + \sum_{j \in I} w_{j \to i} X_t(j) & \text{otherwise} \end{cases} \tag{8}$$

and we see that the knowledge of the $\{V_t(i)\}_{i \in I}$, the $\{X_t(i)\}_{i \in I}$ and the $\{U_t(i)\}_{i \in I}$ is enough to get both the $\{V_{t+1}(i)\}_{i \in I}$ and the $\{X_{t+1}(i)\}_{i \in I}$, a nice Markovian setting.

Remark 2 If $\xi = 0$ and $d_{j \to i} \leq d_{max}$ for all $(i,j) \in I^2$ then Eq. 6 is still "simple" but not as simple as Eq. 8, namely:

$$V_{t+1}(i) = \begin{cases} 0 & \text{if } U_t(i) \leq \phi_i\left(V_t(i)\right) \\ \rho V_t(i) + \sum_{j \in I} w_{j \to i} X_{t-d_{j \to i}}(j) & \text{otherwise} \end{cases} \tag{9}$$

and the the knowledge of the $\{V_t(i)\}_{i\in I}$, the $\{X_t(i)\}_{(t,i)\in\{t-d_{max},\dots,t\}\times I}$ and the $\{U_t(i)\}_{i\in I}$ is enough to get both the $\{V_{t+1}(i)\}_{i\in I}$ and the $\{X_{t+1}(i)\}_{i\in I}$, a not so nice but still Markovian setting.

Remark 3 In the general case where $\xi > 0$, the Markovian setting is lost since we might need to go back an infinite number of time steps in the past in order to compute $V_{t+1}(i)$.

Remark 4 In equation (6), one has to always go back to $L_t(i)$ on every evaluation of $V_t(i)$. This would be computationally inefficient. If we add an auxiliary variable to track only the synaptic inputs (lets call it $S_t^{syn}(i)$), eq. (6) can be extended to the following set of equations:

$$V_{t+1}(i) = \begin{cases} 0 & \text{if } U_t(i) \le \phi_i\left(V_t(i)\right) \\ \rho V_t(i) + \zeta\left(S_t^{syn}(i) + S_t^{ext}(i)\right) & \text{otherwise} \end{cases} \tag{10}$$

and

$$S_{t+1}^{syn}(i) = \begin{cases} 0 & \text{if } U_t(i) \le \phi_i\left(V_t(i)\right) \\ \xi S_t^{syn}(i) + \sum_{j\in I} w_{j\to i} X_{t+1-d_{j\to i}}(j) & \text{otherwise} \end{cases} \tag{11}$$

where ζ is a conversion factor from synaptic current to membrane potential.

Remark 5 I'm still thinking how delay should be correctly implemented in eq. (11). Although it was discussed that $X_{t-d_{j\to i}}(j)$ should be used, discretization method introduced in Rotter and Diesmann (1999) suggests the use of $X_{t+1-d_{j\to i}}(j)$. The difference should be negligible, but which one is more likely to be correct?

Remark 6 How to correctly implement refractory period (τ_{ref}), the time during which neuron remains silent after emmiting a spike? To a first approximation I will implement following the method applied in the PD model, but they just keep the membrane potential at zero, but do not reset synaptic current. The discussion on the meeting was that synaptic current should also be reset!

Remark 7 τ_{ref} was not considered in eqs. $(6-9)$, as this was not discussed on the previous meeting.

2 Codes

The coming examples assume that `Python 3` is used.

2.1 Spike trains representation

2.1.1 Binary sequences as unsigned integers

A spike train *realization* is a sequence of 0 and 1 like: 00100101000010001. It can also be viewed as a non-negative (unsigned) integer, assuming that K time steps are considered, $\{x_i = 0, x_{i+1} = 0, x_{i+2} = 1, \dots, x_{i+K} = 1\}$ can be mapped to the integer:

$$S = x_i 2^{K-1} + x_{i+1} 2^{K-2} + \cdots + x_{i+K} 2^0\,.$$

A nice feature of integer representation in Python is that: Integers have unlimited precision; stated differently integers are memory (RAM size) limited. This is a very specific feature of Python, most languages impose a limit (upper bound) to (unsigned) integers, typically $2^{32} - 1$ of $2^{64} - 1$. A binary sequence like the one we just wrote, 00100101000010001, can be directly converted into a integer with:

```
int(0b00100101000010001)
```

```
18961
```

Notice the 0b prefix, that's the way to declare a binary literal in Python, see the int function documentation. Function bin allows us to go the other way around:

```
bin(18961)
```

```
'0b100101000010001'
```

Notice here that the sequence starts (necessarily) with a 1 on the left (after the 0b prefix). This implies that if we want to represent our spike train realizations with (unsigned) integers we must keep track of our sequence lengths in order to add extra 0 on the left side if necessary..

2.1.2 Adding an event (spike or no spike) to a sequence

Adding a "no spike" event to a sequence is trivial we just multiply it by 2 or we shift it one bit to the left:

```
bin(18961 << 1)
```

```
'0b1001010000100010'
```

To add a "spike" event we shift one bit to the left and add 1:

```
bin((18961 << 1)+1)
```

```
'0b1001010000100011'
```

2.1.3 Finding if there was a spike 5 steps ago

If we want to know if there was a spike 5 steps ago, we make a right shift of 5 bit before making a bitwise and with 1:

```
(18961 >> 5) & 1
```

```
0
```

If we try 9 steps ago we get:

```
(18961 >> 9) & 1
```

```
1
```

2.2 Simulating membrane dynamics with synaptic input (PD model case)

2.2.1 Simulating the stochastic PD model: discrete version

We first consider a discrete simulation with time step of 0.1 ms ($d_t = 0.1$ ms), which is the same of the original PD model simulated in NEST.

Under this condition ($d_t = 0.1$ ms) and considering the following parameter values: $\tau_m = 10$ ms, $C_m = 250$ pF, $\tau_{syn} = 0.5$ ms, the set of equation to simulate is:

$$V_{t+1}(i) = \begin{cases} 0 & \text{if } U_t(i) \leq \phi_i\left(V_t(i)\right) \\ \rho V_t(i) + S_t^{syn}(i) + S_t^{ext} & \text{otherwise} \end{cases} \tag{12}$$

$$S_{t+1}^{syn}(i) = \begin{cases} 0 & \text{if } L_t(i) \leq \tau_{ref} \\ \xi S_t^{syn}(i) + \sum_{j \in I_{exc}} \zeta w_{j\to i} X_{t-d_{j\to i}}(j) - \sum_{j \in I_{inh}} g \zeta w_{j \to i} X_{t-d_{j\to i}}(j) & \text{otherwise} \end{cases} \tag{13}$$

where I_{exc} is the set of excitatory neurons, I_{inh} the set of inhibitory neurons, $g = 4$ (the ratio between inhibitory and excitatory synaptic weight), $\rho = e^{-d_t/\tau_m} \approx 0.99$, $\xi = e^{-d_t/\tau_{syn}} \approx 0.82$, $\zeta = \left(e^{-d_t/\tau_m} - e^{-d_t/\tau_{syn}}\right) / \left(1/\tau_{syn} - 1/\tau_m\right) C_m \approx 0.00036$ ms/pF and $\tau_{ref} = 2.0$ ms. Moreover, $w_{j\to i} = \mathcal{N}\left(\mu = 87.8, \sigma = 8.8\right)$ pA and $d_{j\to i} = \mathcal{N}\left(\mu = 1.5, \sigma = 0.75\right)$ ms, and $\zeta^{syn}\mu_{w_{j\to i}} \approx 0.032$ mV ($\mu_{w_{j\to i}}$ is the mean value of $w_{j\to i}$).

Remark 1 When the considered discrete time step is changes, parameters ρ, ξ and ζ have to be changed accordingly.

The firing probability function $\Phi(V_t(i))$ The firing probability function ($\Phi(V_t(i))$) was determined so that the firing rate of the neuron to constant current input (FI-curve) reproduces that of the LIF neuron of the original PD model.

We consider the following function for $\Phi(V)$

$$\Phi(V) = \begin{cases} 0 & \text{if } V \leq V_{rheo} \\ [\gamma(V - V_{rheo})]^r & \text{if } V_{rheo} < V < V_{sat} \\ 1 & \text{if } V \geq V_{sat} = V_{rheo} + 1/\gamma \end{cases} \quad (14)$$

From an implementation point of view, since firing condition is determined by:

$$X_t(i) = \begin{cases} 1 & \text{if } \Phi(V_t(i)) \geq U_i \\ 0 & \text{otherwise,} \end{cases} \quad (15)$$

where U_i is a random number drawn from a uniform distribution in [0,1), eq. (14) can be simplified to:

$$\Phi(V) = \begin{cases} 0 & \text{if } V \leq V_{rheo} \\ [\gamma(V - V_{rheo})]^r & \text{otherwise} \end{cases} \quad (16)$$

Fig. 1 represents the FI-curve of the original LIF neuron (red) and that of the GL model (blue). The FI-curve for the GL neuron was obtained with $\gamma = 0.1$ mV^{-1} (i.e. there is a window of 10 mV for $0 < \Phi(V) < 1$), $r = 0.4$ and $V_{rheo} = 15$ mV, which is represented in Fig. 2.

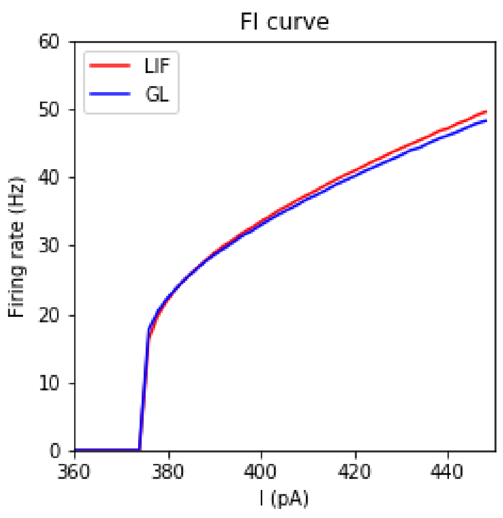

Figure 1: FI curve of LIF (red) and GL (blue) neuron model.

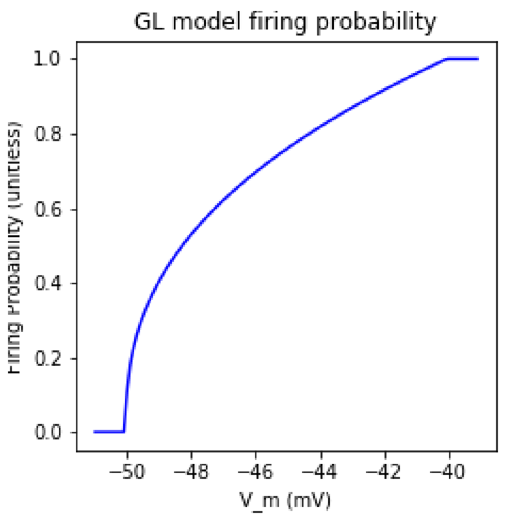

Figure 2: Firing rate ($\Phi(V)$) of the GL model.

Implementation in python For the python implementation try to stick at most to numpy packages, and try to avoid packages as scipy and others. On the code bellow, np stands for numpy as the numpy module is loaded as follow. We also specify the seed for the random number generator:

Remark 1 Poissonian external input is not implemented yet on this version. The code will be updated accordingly, and reported when done.

```
import numpy as np
# random number generator depends on numpy version
# Here I used numpy 1.19:
self.rng = np.random.default_rng(seed=1234)
self.poisson_rng = np.random.default_rng(seed=2345)
# create an empty list to store spike activity
spk_neuron = []
spk_time = []
```

1. Creates an array containing N neurons. The number of neurons is the sum of number of neurons per layer in table 1:

   ```
   V_m = np.zeros(N)
   ```

2. parameters ρ, ξ and ζ are calculated using neuron parameters described above:

   ```
   rho = np.exp(-dt/tau_m)
   xi = np.exp(-dt/tau_syn)
   zeta = ( np.exp(-dt/tau_m) - np.exp(-dt/tau_syn_ex) ) / ( 1/tau_syn - 1/tau_m )*C_m
   ```

   Store connectivity data as python dictionary (dictionary is an efficient way of accessing data):

   ```
   conn_dict = {pre_id:{'target':[array of target neuron id],
                        'weight':[array of synaptic weights],
                        'delay':[array os synaptic delays]}
   ```

3. evolve time (t) and $V_t(i)$ are evaluated as:

   ```
   t = 0.0
   dt = 0.1
   while t<= t_sim:
       t = t + dt
       V_m = rho*V_m + zeta*I_syn
   ```

   where summed_weights is a 2D array containing the sum of synaptic inputs arriving at the neuron. the first index represents synaptic delays in step counts. Therefore, index 0 represents the effective synaptic inputs, that is, spikes that occurred d steps in the past, with d representing the delay. This 2D array is than shifted along the delay axes to accommodate the next *effective* spike inputs.

4. Draw Poissonian spike inputs

```
# generate poisson spike train for time window dt
# convert time from ms to s (poisson_rate is in Hz, while dt is in ms)
lambda_ = poisson_rate * dt * 1e-3
poisson = poisson_rng.poisson(lambda_, N) # draw sample from a poisson distribution
```

5. evolve $I_syn(i)$ as:

```
I_syn = xi * I_syn_ex + summed_weights[0] + poisson*weight
```

   where summed_weights is a 2D array containing the sum of synaptic inputs arriving at the neuron. the first index represents synaptic delays in step counts. Therefore, index 0 represents the effective synaptic inputs, that is, spikes that occurred d steps in the past, with d representing the delay. This 2D array is than shifted along the delay axes to accommodate the next *effective* spike inputs.

6. Find for neurons in refractory period and reset its membrane potential and I_syn

```
idx_ref = np.where(is_ref > 0)[0]
V_m[idx_ref] = V_reset
I_syn_ex[idx_ref] = 0.0
I_syn_in[idx_ref] = 0.0
```

   is_ref is a counter that stores the number of simulation step necessary for the refractory period. Each time step this counter is decreased by one, therefore, when this counter is ≤ 0, indicates that the neuron is no more in its refractory period.

7. calculate the firing probability function ($\phi(V)$):

```
V_diff = V_m - V_rheo
idx_neg = np.where(V_diff < 0)
V_diff[idx_neg] = 0.0
phi = np.power(gamma*V_diff, r)
```

   where V_rheo is the rheobase potential, the minimum potential necessary for a neuron to spike, gamma=0.1 and r=0.4. V_diff is clipped at zero to force all potentials below V_rheo to have a firing probability of zero.

8. draw an array of uniform random numbers, with array size equal to neuron population size, and choose which neuron has spiked.

```
# if using python > 1.17:
# U_i = self.rng_.random(self.N) # needs numpy > 1.17
# I use numpy 1.16.2:
U_i = np.random.uniform(size=self.N)
# get index of neurons that spiked
idx_spiked = np.where(phi >= U_i)[0]
```

9. set refractory count of the neurons that spiked

```
is_ref[idx_spiked] = refractory_count
```

10. send spike event to postsynaptic neurons.synaptic weight is then sent to the target neuron as follow (spike activity is also stored to a variable and/or file):

```
for pre in idx_spiked:
    for idx, post in enumerate(conn_dict[pre+1]['target']):
        idx_delay = int(conn_dict[pre+1]['delay'][idx]/dt) # delay step counts
        summed_weights[idx_delay][post-1] += conn_dict[pre+1]['weight'][idx]
    spk_neuron.append(pre+1)
    spk_times.append(t)
```

Remark 1 note that idx_spike starts at zero, whereas neuron_id starts at 1.

Remark 2 If delays are all the same between pairs of neurons, the inner loop may be eliminated as:

```
for pre in idx_spiked:
    summed_weights[idx_delay][conn_dict[pre+1]['target']-1] += conn_dict[pre+1]['weight']
    spk_neuron.append(pre+1)
    spk_times.append(t)
```

11. shift summed_weight array discarding the transmitted weights, and adding zeros to the end of the array

```
for idx in range(1,len(summed_weights)):
    summed_weights[idx-1] = summed_weights[idx]
summed_weights[-1][:] = 0.0
```

Remark 3 Steps 3 – 9 are within the same loop, and closes one simulation step.

Implementation in C++ – Nilton ToDo: describe de C++ code by Nilton

Implementation in C – Stolfi ToDo: Describe de C code by Stolfi

2.3 Analysis of the GL model

2.3.1 The firing probability curve revisited – comparison with Stolfi's code

ToDo: describe Stolfi's code somewhere.

Stolfi's code uses simulation time step (dt) of 1 ms. Fig. 3 shows the FI-curve of the implemented GL neuron (C++ version) to a constant current input ranging from 300 – 9000 pA varied in 100 pA steps, simulated for 10 seconds. Blue dots represents the FI-curve obtained by a dt of 0.1 ms, and orange dots by a dt of 1 ms.

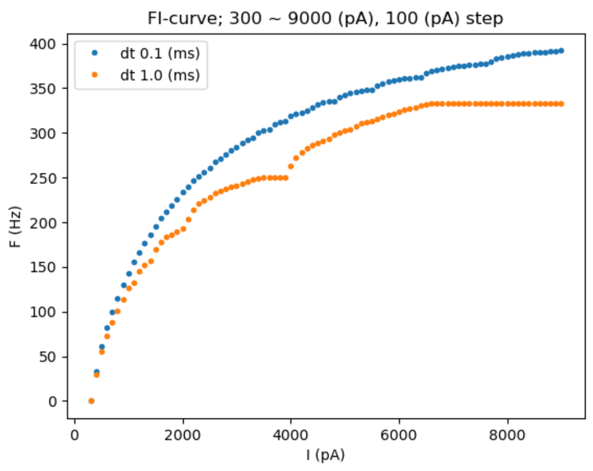

Figure 3: FI-curve

Note that at dt of 1 ms, the firing rate saturates at about 333 Hz. This is due to the fact that the neuron has a refractory period (τ_{ref}) of 2 ms. In such case, when a neuron fires at time t, it will remain silent until time $t + \tau_{ref}$, and the closes next spike will be at time $t + \tau_{ref} + 1$, resulting in a period of 3 ms which is about 333 Hz. This result

On the other hand, at dt of 0.1 ms, the FI curve should saturate at nearly 500 Hz, as the shortest period will be 2.1 ms.

2.3.2 Firing rate of a single neuron the the layer specific input

The GL neuron was submitted to the layer specific external input shown in section **??**.

Table 5: Firing rate to layer specific constant input

		firing rate (Hz)	
layer	DC (pA)	dt=0.1	dt=1
L23E	561.92	74.4	65.4
L23I	526.8	67.0	60.1
L4E	737.52	105.7	91.6
L4I	667.28	94.1	82.2
L5E	702.4	100.1	88.9
L5I	667.28	94.2	82.8
L6E	1018.48	145.4	127.4
L6I	737.52	105.6	91.6

Table 6: Firing rate to layer specific constant input – Stolfi version (dt=1(ms))

layer	I_{avg}	I_{dev}	firing rate (Hz)
L23E	2.221	0.637	79.84
L23I	2.115	0.627	69.86
L4E	2.943	0.717	119.76
L4I	2.678	0.638	109.78
L5E	2.813	0.724	109.78
L5I	2.636	0.686	99.80
L6E	4.063	0.858	177.64
L6I	2.990	0.711	121.76

2.4 Analysis of the PD model

2.4.1 Outdegree

Outdegree is the number of outgoing synapses. Fig. 4 shows the histogram of outdegrees in a realization of the PD model (NEST config: master_seed=55; number of virtual processes=16)

An example of maximum and minimum connection is shown in table 7.

Table 7: Maximum and minimum outdegrees per layer

	max	min
L23e	10414	4
L23i	16270	51
L4e	24829	19
L4i	25087	127
L5e	35869	67
L5i	31656	72
L6e	20025	0
L6i	32872	19

3 Pseudo Random Number Generator (PRNG)

The pseudo random number generation is going to be done with the Xoroshiro128+ algorithm using a C++ wrapper implemented by Matthias Rahlf of the original C implementation of Blackman and Vigna. To obtain the different distributions (*i.e.* real uniform [0, 1), ingeter uniform [min, max), normal distribution (μ = 0, \$$\sigma$=1) and poisson distribution (λ)), the standard library will be used.

3.1 A test case of the PRNG

Here the generation of 8×10^8 uniformly distributed real numbers in the range [0, 1) will be drawn, and the time of execution measured for the cases of xoroshiro128+ and mt19937 (Mersenne Twister) generators, where the latter is present in the standard libray. the number of random number generated is approximatly the number of random numbers that will be necessary for simulating the PD microcircuit model for 10^4 steps.

3.1.1 The xoroshiro128+ case

The code for drawing 8×10^8 uniformly distriburted random number with the xoroshiro128+ generator:

```cpp
#include "xoroshiro128plus.hpp"
#include <random>

xoroshiro128plus prng;
std::uniform_real_distribution<> udist;

int N=800000000;

int main() {
  double val;

  // seed half the maximal value
  prng.seed(prng.max()*0.5);

  for (int i=0; i<N; i++) {
    val = udist(prng);
  }
}
```

Compiling the source code and measuring the execution time:

```
g++ ./src/test_prng_xoroshiro128plus.cpp -o ./src/run_test_prng_xoroshiro128plus
time ./src/run_test_prng_xoroshiro128plus
```

```
real 0m25.935s
user 0m25.911s
sys 0m0.000s
```

3.1.2 The mt19937 case

The code for drawing 8×10^8 uniformly distriburted random number with the mt19937 generator:

```
#include <random>

std::mt19937 prng;
std::uniform_real_distribution<> udist;

int N=800000000;

int main() {
  double val;

  // seed half the maximal value
  prng.seed(prng.max()*0.5);

  for (int i=0; i<N; i++) {
    val = udist(prng);
  }
}
```

Compiling the source code and measuring the execution time:

```
g++ ./src/test_prng_mt19937.cpp -o ./src/run_test_prng_mt19937
time ./src/run_test_prng_mt19937
```

```
real 0m0.060s
user 0m0.058s
sys 0m0.001s
```

3.1.3 Which is faster?

For a very simple code that just generates uniformly distributed random numbers, the xoroshiro128+ algorithm was about 33% faster than the widely used mt19937 generator.

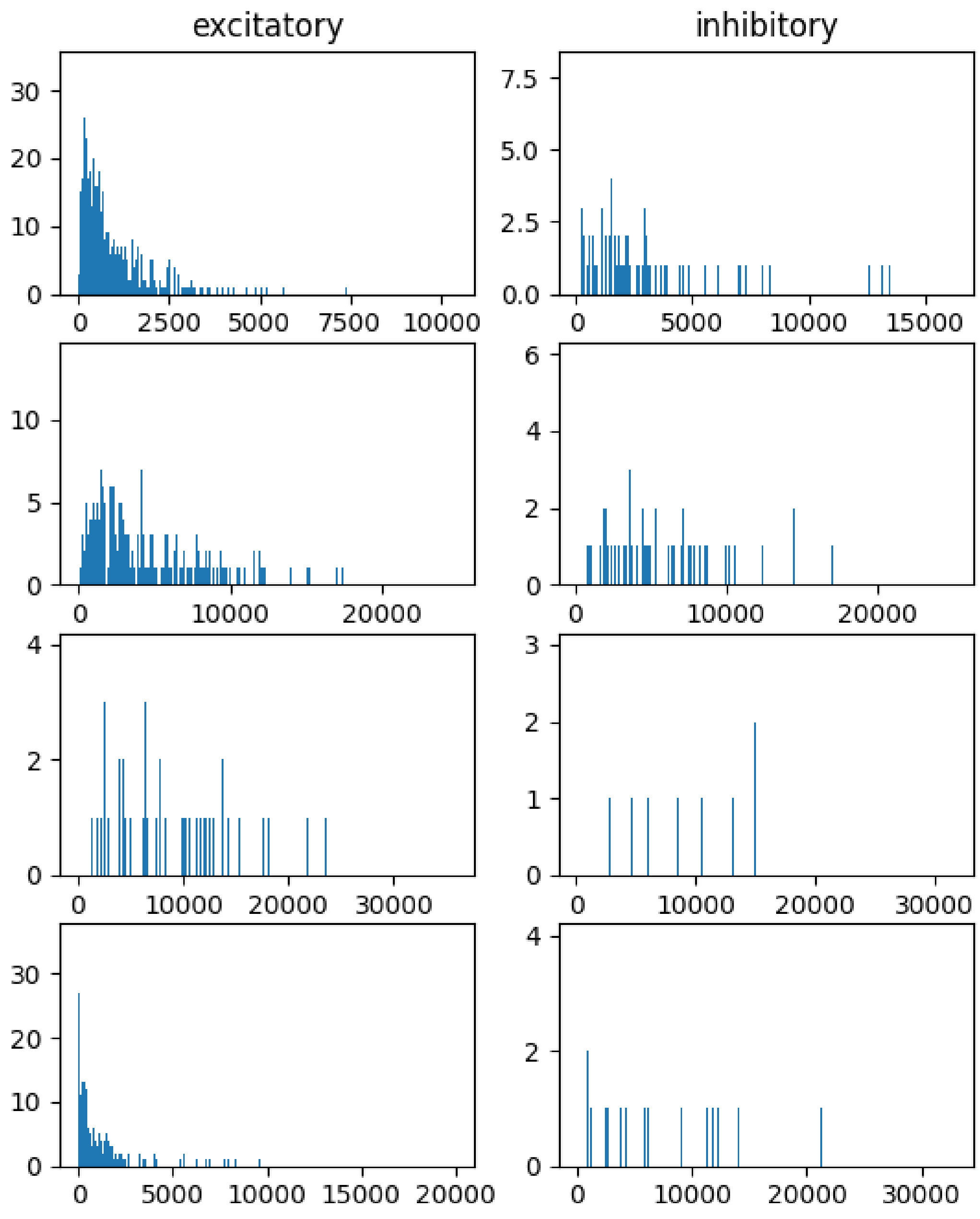

Figure 4: Caption