Implementation of the Density Evolution method with Fokker-Plank for L-NLIF Model in Python

Valentin Haenel

January 28, 2009

1 Model

The original model is a L-NLIF model, which consists of a Linear (L) Filter, followed by a probabilistic or Noisy (N) form of Leaky Integrate and Fire spike generation(LIF) (Paninski et al.[1]). For sake of completeness the model is briefly described here.

The evolution of the Voltage V is given by:

$$dV = (-g(V(t) - V_{leak}) + I_{stim}(t) + I_{hist}(t))dt + W_t \tag{1}$$

Where g is the leak conductance V_{leak} is the leak reversal potential I_{stim} is the convolution of linear filter with the input signal, I_{hist} is the spike current history and W_t is standard Gaussian white noise. More precisely:

$$I_{stim}(t) = \vec{k}\vec{x}(t) \tag{2}$$

and

$$I_{hist}(t) = \sum_{j=0}^{i-1} h(t - t_j) \tag{3}$$

Where k is a linear filter and h is a fixed postspike current waveform.

2 Density Evolution

If we define

$$P(V,t) \equiv P(V(t) \cap V(s) < V_{th} \forall s < t) \tag{4}$$

This refers to the probability of $V(t)$, the membrane potential, being less than V_{th} the firing threshold until time t.

We need to solve the following Fokker-Planck drift diffusion equation numerically.

$$\frac{\partial P(V,t)}{\partial t} = \frac{\sigma^2}{2}\frac{\partial^2 P(V,t)}{\partial V^2} + g\frac{\partial[(V - V_{rest})P(V,t)]}{\partial V} \tag{5}$$

Given the boundary conditions:

$$P(V_{th}, t) = 0 \tag{6}$$

$$P(V, 0) = \delta(V - V_{reset}) \tag{7}$$

when solving numerically we also need a lower bound on the voltage V_{lb}, so this becomes an additional boundary condition:

$$P(V_{lb}, t) = 0 \tag{8}$$

Ideally $V_{lb} = -\infty$

And the definition of V_{rest}.

$$V_{rest}(t) = V_{leak} + \frac{1}{g}(\vec{k} \cdot \vec{x}(t) \sum_{j=0}^{i-1} h(t - t_j)) \tag{9}$$

We can rewrite this as:

$$\frac{\sigma^2}{2}\frac{\partial^2 P(V,t)}{\partial V^2} + g(V - V_{rest})\frac{\partial P(V,t)}{\partial V} + gP(V,t) - \frac{\partial P(V,t)}{\partial t} = 0 \tag{10}$$

Since we know that:

$$\frac{\partial[(V - V_{rest})P(V,t)]}{\partial V} = \frac{\partial(V - V_{rest})}{\partial V}\dot{P} + \frac{\partial P}{\partial V}\dot{}(V - V_{rest}) \tag{11}$$

and

$$\frac{\partial(V - V_{rest})}{\partial V} = 1 \tag{12}$$

Next we discretize time and potential. We adopt the notation that Potential is discretized into W intervals of length w and indexed by $\nu = 0, 1, \ldots W$. Time is discretized into U intervals of length u and indexed by: $\tau = 0, 1, \ldots U$.

$P_{\nu,\tau} = P(\nu w, \tau u)$

Before we can write down the computationally stable Crank-Nicolson method we must first write down our finite differencing scheme:

$$\hat{P} = P_{\nu,\tau} \tag{13}$$

$$\frac{\partial \hat{P}}{\partial t} = \frac{P_{\nu,\tau+1} - P_{\nu,\tau}}{u} \tag{14}$$

$$\frac{\partial \hat{P}}{\partial V} = \frac{P_{\nu+1,\tau} - P_{\nu-1,\tau}}{2w} \tag{15}$$

$$\frac{\partial^2 \hat{P}}{\partial V^2} = \frac{P_{\nu+1,\tau} - 2P_{\nu,\tau} + P_{\nu-1,\tau}}{w^2} \tag{16}$$

Using the Crank-Nicolson scheme we may now rewrite the derivatives using a new finite differencing scheme which is centered around $t + u/2$. Bearing in mind that this is only an approximation we get:

$$P_{CN} = \frac{P_{\nu,\tau} + P_{\nu,\tau+1}}{2} \tag{17}$$

$$\frac{\partial P_{CN}}{\partial t} = \frac{P_{\nu,\tau+1} - P_{\nu,\tau}}{u} \tag{18}$$

$$\frac{\partial P_{CN}}{\partial V} = \frac{P_{\nu+1,\tau} + P_{\nu+1,\tau+1} - P_{\nu-1,\tau} - P_{\nu-1,\tau+1}}{4w} \tag{19}$$

$$\frac{\partial^2 P_{CN}}{\partial V^2} = \frac{P_{\nu+1,\tau} - 2P_{\nu,\tau} + P_{\nu-1,\tau} + P_{\nu+1,\tau+1} - 2P_{\nu,\tau+1} + P_{\nu-1,\tau+1}}{2w^2} \tag{20}$$

if we now let:

$$\begin{aligned} a &= \frac{\sigma^2}{2} \\ b &= g(V - V_{rest}) \\ c &= g \end{aligned}$$

and multiply throughout with $4w^2u$

we may rearrange all the $P_{*,\tau+1}$ terms on the left hand side:

$$\overbrace{-(2au+bwu)}^{A_\nu} P_{\nu+1,\tau+1} + \overbrace{(4au-2cw^2u+4w^2)}^{B_\nu} P_{\nu,\tau+1} \overbrace{-(2au-bwu)}^{C_\nu} P_{\nu-1,\tau+1} = \underbrace{(2au+bwu)P_{\nu+1,\tau} + (-4au+2cw^2u+4w^2)P_{\nu,\tau} + (2au-bwu)P_{\nu-1,\tau}}_{D_\nu} \tag{21}$$

For each $\nu = 1, \dots, W-1$

We note here that we have obtained $W-1$ simultaneous equations which we may now rewrite in the following tridiagonal matrix notation.

$$\underbrace{\begin{pmatrix} B_0 & A_0 & 0 & 0 & \cdots & 0 \\ C_1 & B_1 & A_1 & 0 & \cdots & 0 \\ 0 & C_2 & B_2 & A_2 & 0 & \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & C_{W-1} & B_{W-1} & A_{W-1} \\ 0 & \cdots & & 0 & C_W & B_W \end{pmatrix}}_{\Lambda} \underbrace{\begin{pmatrix} P_{0,\tau+1} \\ P_{1,\tau+1} \\ P_{2,\tau+1} \\ \vdots \\ P_{W-1,\tau+1} \\ P_{W,\tau+1} \end{pmatrix}}_{\chi} = \underbrace{\begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ \vdots \\ D_{W-1} \\ D_W \end{pmatrix}}_{\beta} \tag{22}$$

However this matrix equation has W+1 equations, so we must use the boundary conditions to specify the first and last rows of Λ. We can then use a tridiagonal matrix algorithm to solve $\Lambda\chi = \beta$.

This allows us to iteratively obtain the density evolution of the membrane potential for a given time interval.

The maximum likelihood optimizer proposed in [1] however uses the first passage time density at a given time point which is defined as:

$$fpt(t) = -\frac{\partial}{\partial t} \int P(V,t)dV \tag{23}$$

and this can be easily computed from an $P(V,t)$ matrix by summing across P and finding differences across t.

The final likelihood is then simply

$$L_{\vec{x},t_i} = \Pi_i fpt(t_i) \tag{24}$$

The pseudocode for the objective function can be described as follows:

for each spike interval

1. compute $P(V,t)$ as matrix
2. compute fpt as vector
3. compute the product of the last scalar in the fpt vector

3 Implementation

3.1 Availability and Design

The algorithm has been implemented in Python, and the full source code has been made available within a revision control system under a free software license at: http://github.com/esc/molif/tree/master.

The software is structured into a series of modules, one for the neuron model, one for the pde solver, one to minimize the negative likelihood, one for monte carlo simulation and a utility module.

The PDE solver uses sparse matrix algorithms to store and solve the tridiagonal matrix. Ideally we would like to use the routine $TRIDAG$ described in [2], however an open source implementation in python was not available.

References

[1] L. Paninski, J. W. Pillow, and E. P. Simoncelli. Maximum likelihood estimation of a stochastic integrate-and-fire neural encoding model. *Neural Comput*, 16(12):2533–2561, Dec 2004.

[2] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, UK, 2nd edition, 1992.