Low-Fidelity Neurons for Large-Scale Digital Simulations

January 31, 2013

[TODO: table of population-level and neuron-level properties]

1 Response Diversity

Note: a wide range of intercepts leads to a difficult distribution of scales and biases. Most numbers are small (e.g. <10) while a few are much larger (e.g. >10,000). Both can't be represented accurately with small fixed-point numbers. The problem is reduced with a narrowed range of intercepts (e.g. 90% represented range). Probably some combination of representation as exponents and range capping is needed. Code for plotting the problem in Nemo is given below.

```
sg = LIFSpikeGenerator(.001, .002, .02,
    -.9+1.8*rand(1,1000), 100+100*rand
    (1,1000), 0)
scatter(sg.scales, sg.biases)
```

2 Reference Model: Leaky-Integrate-and-Fire

This is a simple point model of spiking due to Lapique (1907) in which subthreshold response to driving current I is modelled as a low-pass filter as

$$\tau_{RC}\frac{dV}{dt} = -V + I.$$

When the membrane current V reaches spike threshold V_{th}, the neuron is considered to spike, but the details of the membrane potential trajectory are not modelled. V is then held at a reset potential V_0 for a short refractory period τ_{ref} before subthreshold integration resumes. We typically normalize V so that $V_0 = 0$ and $V_t h = 1$.

With a typical time step such as 1ms, the firing rate may be severely quantized at high values for near-constant input. For example, if threshold crossing would occur in 3.4ms of continuous time, it would be detected at 4ms, and (because V is reset after each spike) the cell would fire at 250 spikes/s, which is quite different than the correct value of 294 spikes/s. This problem can be reduced by interpolating between samples when a spike occurs and allowing integration over the appropriate fraction of the time step after the refractory period.

```
% An implementation of our standard LIF model. Run an example simulation
% like this:
%
% drive = -1:.002:1; %ramp input
% n = 50; # of neurons
% tauRef = .002; tauRC = .02;
% intercepts = -1+2*rand(1,n);
% maxRates = 100+100*rand(1,n);
% scales = ((1 ./ (1 - exp( (tauRef - (1 ./ maxRates)) / tauRC))) - 1) ./ (1 - intercepts);
% biases = 1 - scales .* intercepts;
% spikes = referenceLIF(.001, drive, .002, .02, scales, biases);
% imagesc(spikes), colormap('gray'), xlabel('Time (ms)'), ylabel('Neuron #')

% dt: time step (s)
% drive: input in the range [-1,1] (1 X #steps)
% tauRef: spike refractory period
% tauRC: membrane time constant
% scales: list of input scale factors for different neurons (1 X #neurons)
% biases: list of additive input biases for different neurons (1 X #neurons)
function spikes = referenceLIF(dt, drive, tauRef, tauRC, scales, biases)
    V = zeros(size(scales));
    lastSpike = -1*ones(size(scales));
    spikes = zeros(length(scales), length(drive));

    for i = 1:length(drive)
        endTime = dt*i;
        intTimes = min(dt, max(0, endTime - lastSpike - tauRef));

        current = biases + scales .* drive(i)';
        dV = (current - V) ./ tauRC;

        V = V + intTimes .* dV;
        V = max(0, V);

        spikeIndices = find(V >= 1);

        % here we are interpolating the spike time between time steps (important at high rates)
        lastSpike(spikeIndices) = endTime - (V(spikeIndices)-1) ./ dV(spikeIndices);

        V(spikeIndices) = 0;
        spikes(spikeIndices,i) = 1;
    end
end
```

3 Model One: Random Decay

The LIF model can be implemented with V as an unsigned integer value with few bits (perhaps 7 or 8 bits), normalized from zero to the highest (e.g. 127 or 255).

This variant does not accurately model small rates of change in the membrane potential such that $|\Delta V| < 1$ (in a time step of length Δt). The problem is obvious during decay to 0. If a step of < 1 is rounded to zero, then V decays to $\tau_{RC}/\Delta t$ and remains constant thereafter rather than decaying gradually to zero. If it is rounded to -1, V approaches 0 rapidly. This problem can be reduced by decaying probabilistically as

$$\Delta V \rightarrow \begin{cases} 0 & if\ r > -\Delta V \\ -1 & otherwise \end{cases}$$

```
% testing discrete low-pass filter

steps = 1000;
x = uint16(zeros(1,steps));
input = zeros(1,steps);
input(1:100:end) = 150;
y = zeros(1,steps);
x(1) = input(1);
y(1) = input(1);
alpha = 255;
for i = 2:steps
%     foo = uint16(x(i-1)) * uint16(alpha);
%     x(i) = bitshift(foo, -8);

    bar = uint16(x(i-1)) * uint16(256) - uint16(x(i
        -1)) * uint16(alpha);
    decrement = bitshift(bar, -8);

    r = uint8(255*rand);

    if r < mod(bar, 2^8)
        decrement = decrement + 1;
    end
    x(i) = min(255, x(i-1) - decrement + input(i));

    y(i) = y(i-1) * alpha/256 + input(i);
end

figure, hold on, plot(x, 'r.'), plot(y, 'g')
```

4 Model Two: Bit Shift

This is a very simple model that is less directly related to biophysics, but has the major functional properties needed for a large-scale model.

The useful input current ranges from 0 to 600 and gives firing rates up to about 120Hz. The model has two parameters: RC_{bits} (which kind of acts like τ_{RC}, in that it affects the linearity of the tuning curve) and the reset value which is the refractory time. For each neuron, the total amount of saved state is 10 bits (8 bits for self.voltage and 2 bits for self.reset). [TODO: Terry has ideas for making the response smoother but doesn't think the added complexity is beneficial]

4.1 Python Code

```
class FixedNeuron:
Â Â def __init__(self):
Â Â Â Â self.voltage=0
Â Â Â Â self.spike=False
Â Â Â Â self.reset=0
Â Â def tick(self,current):
Â Â Â Â self.spike=False
Â Â Â Â if self.reset>0:
Â Â Â Â Â Â self.reset-=1
Â Â Â Â Â Â return

Â Â Â Â rc_bits=3
Â Â Â Â dv=(current-self.voltage)>>rc_bits

Â Â Â Â self.voltage+=dv
Â Â Â Â if self.voltage<0: self.voltage=0

Â Â Â Â if self.voltage>255:
Â Â Â Â Â Â self.voltage-=256
Â Â Â Â Â Â self.reset=3
Â Â Â Â Â Â self.spike=True
```

5 Model Three: Log Decay

This is a integer-valued variant of the LIF model that does not need a random number generator, but has to calculate a logarithm instead. The decay rounding problem is addressed by using a new state variable y that is related to the normal filter state variable x, but decays at a constant rate. Specifically, $y = (\tau/\Delta t)\log(x)+255$. This decays nicely but adds a new wrinkle in that the input has to be scaled nonlinearly (see Matlab code below).

5.1 Matlab Code

```matlab
% Testing similarity of exponential decay vs. linear decay with
% exponential input.

dt = .001; %time step (s)
T = .25; %length of run (s)
time = dt:dt:T;
tau = .02; %filter time constant

u = .06*ones(1,length(time)) + .05*randn(1,length(time)); %random input

x = zeros(1,length(time)); %filter state variable
for i = 2:length(x)
    x(i) = x(i-1) - dt*x(i-1)/tau + u(i-1);
    if x(i) >= 1
        x(i) = 0;
    end
end
plot(time, x)

peak = 255; %top of byte range
y = zeros(1,length(time), 'uint8'); % note: y is a function of x that decays at a constant rate with zero input, specifically y = (tau/dt) log(x) + peak
for i = 2:length(y)
    y(i) = y(i-1) -1 + tau / dt / (exp(dt/tau*(double(y(i-1))-peak)) + u(i-1)/5) * u(i-1);
    if y(i) < 0 || y(i) >= peak
        y(i) = 0;
    end
end

z = exp(dt/tau*(double(y)-peak)); % z ~= x
hold on, plot(time, z, 'r.')

legend('double', 'uint8')
```