The Integrate-and-Fire Model

The integrate-and-fire neuron is one of the simplest models of a neuron's electrical properties and probably the most commonly used in the field of neuroscience. The essence of the model is to divide the voltage changes of the neuron into two parts:

1) Below threshold, it is assumed that the membrane behaves passively (i.e. has no voltage-dependent ion channels) and acts as a leaky capacitor whose voltage, in the absence of injected current, decays (or "leaks") to a resting level E_L (short for "E_{Leak}").

2) When the voltage reaches the action potential threshold (due to injected currents charging up the membrane), the model assumes that the voltage spikes immediately to a level V_{spike} and is then immediately reset to a hyperpolarized level V_{reset}. There is no explicit modeling of the ion channel kinetics responsible for this spiking. Rather, it is simply assumed (a reasonable assumption…) that once the cell reaches its threshold it will rapidly produce an action potential and reset itself. The reason we can get away with this assumption is that we don't really care about the exact shape of the action potential: since all action potentials sent down the axon are to a good approximation identical, the only informative feature of a neuron's spiking is the *times* at which the action potentials occur.

I. Mathematics of the integrate-and-fire neuron

Consider a neuron modeled as a leaky capacitor with membrane resistance R_m, time constant τ_m=$R_m C_m$ (where C_m is the membrane's capacitance), and resting potential E_L. Below the action potential threshold, the equation for the voltage of this cell when it receives current injection I_e is:

$$\tau_m \frac{dV}{dt} = E_L - V + R_m I_e \qquad (1)$$

Exercise: When the current injection I_e is constant over time, verify
(i) that the solution to this equation is:

$$V(t) = E_L + R_m I_e + (V(t_0) - E_L - R_m I_e)\exp\left(-(t-t_o)/\tau_m\right) \qquad (2)$$

where the constant t_0 is any reference time. (Do this by inserting the solution on both sides of equation (1). Recall that $\frac{d}{dt}\left(e^{f(t)}\right) = e^{f(t)}\frac{df(t)}{dt}$).

(ii) that when t=t_0, the left and right sides of equation (2) agree (and both equal what value?), and

(iii) that when $t \to \infty$, $V(t) \to V_\infty \equiv E_L + R_m I_e$.

Setting t_0 equal to the current time in a computer simulation and t equal to the time a single time-step Δt later gives the one-time-step update rule we will use for simulating the integrate-and-fire neuron (and which we will use more generally for simulating any equation of the form of equation (1) above, e.g. in your homework for the spike-rate-adaptation conductance if we substitute g_{sra} for V and set E_L=I_e=0):

$$V \to E_L + R_m I_e + (V - E_L - R_m I_e)\exp\left(-\Delta t/\tau_m\right),$$

where this notation means that, in the time step Δt, the voltage gets updated from its old value to the value on the right of the arrow, i.e. (in mathematics notation, in terms of the actual time t)

$$V(t+\Delta t) = E_L + R_m I_e(t) + (V(t) - E_L - R_m I_e(t)) \exp\left(-\Delta t / \tau_m\right) \quad (3)$$

or (in MATLAB notation, in terms of values at time step i)

$$V(i+1) = E_L + R_m I_e(i) + (V(i) - E_L - R_m I_e(i)) \exp\left(-\Delta t / \tau_m\right) \quad (4)$$

Now let's see how to implement this…

II. Today's model: the integrate-and-fire neuron

In the following sections, our goal will be to verify the firing rate r vs. (constant) injected current relationship for the integrate-and-fire model neuron:

$$r_{theory} = \frac{1}{t_{isi}} = \begin{cases} \left[\tau_m \ln\left(\frac{R_m I_0 + E_L - V_{reset}}{R_m I_0 + E_L - V_{th}}\right)\right]^{-1}, & \text{if } I_0 > I_{threshold} = \frac{V_{th} - E_L}{R_m} \\ 0, & \text{if } I_0 \le I_{threshold} = \frac{V_{th} - E_L}{R_m} \end{cases} \quad (5)$$

where t_{isi} is the interspike interval for an integrate-and-fire neuron receiving constant current input I_e=I_0=constant and $I_{\text{threshold}}$ is the minimum level of current injection needed to make the neuron fire.

To confirm this relation, we will build a model integrate-and-fire neuron that obeys the equation (equation (1))

$$\tau_m \frac{dV}{dt} = E_L - V + R_m I_e$$

for voltages below the action potential threshold and spikes whenever it reaches the action potential threshold. For concreteness, we will use parameter values E_L = -70 mV, R_m = 10 MΩ, and τ_m=10 ms. We will assume that, initially (i.e. at t=0), V = E_L.

To model the spiking of the neuron when it reaches threshold, we will assume that when the membrane potential reaches V_{th} = -55 mV, the neuron fires a spike and then resets its membrane potential to V_{reset} = -75 mV.

We will inject various levels of current I_e into the neuron and, to calculate the firing rate, we will count the number of action potentials in a fixed amount of time. For starters, we will assume that the neuron receives a 300-ms-long current pulse of magnitude I_0 beginning at time t_{pulse}=100 ms and plot several representative values of I_e that produce firing rates between 1 and 100 Hz. We will run our our simulations for 500 ms total (i.e. 100 ms with I_e=0; 300 ms with I_e=I_0 > 0; and another 100 ms with I_e=0). We will run our simulation with a time step dt = 0.1 ms.

As the final output, we will produce graphs of voltage vs. time for several levels of I_0 and a summary graph comparing r_{theory} to the average firing rate r_{ave} of the neuron over a measurement time T, defined as:

$$r_{ave} = \frac{\text{\# of AP's in time T}}{\text{T}} = \frac{N}{\text{T}} \quad (6)$$

III. Step 1: Model the subthreshold voltage dynamics

As a first step, we will model just the subthreshold dynamics of the model governed by equation (1). In the next section, we will add the spiking. Our general overall strategy is to break our code into the following sections:

1. Define the parameters in the model
2. Define the vectors that will hold our final results such as the time, voltage, and current; and assign their initial values corresponding to t=0.
3. Integrate the equation(s) of the model to obtain the values of the above vectors at later times by updating the values at the previous time step with the update rule.
4. Make pretty plots of our results.

First, let's open a new m-file and name it something memorable like "IntAndFire1.m".
Let's put a comment right at the top:

```
% Lab 2: Build an integrate-and-fire model neuron and observe its spiking
%            for various levels of injected current
```

Now let's ensure (as we should *always* do) that all variables are cleared and figures are closed by adding:

```
clear all;  %clear all variables
close all;  %close all open figures
```

It is good programming style to next assign values to all model parameters in a well-marked section of your code. We need to assign values to the following parameters (type this code):

```
%DEFINE PARAMETERS
dt = 0.1; %time step [ms]
t_end = 500; %total time of run [ms]
t_StimStart = 100;  %time to start injecting current [ms]
t_StimEnd = 400; %time to end injecting current [ms]
E_L = -70; %resting membrane potential [mV]
V_th = -55;  %spike threshold [mV]
V_reset = -75; %value to reset voltage to after a spike [mV]
V_spike = 20; %value to draw a spike to, when cell spikes [mV]
R_m = 10; %membrane resistance [MOhm]
tau = 10; %membrane time constant [ms]
```

Notice that we have made a comment describing each parameter and noting its units. Checking that your units make sense (i.e. that both sides of any equation have the same units) is *very* important and a good way to find errors.

Next, it is good to set up the initial conditions for the run (i.e. specify what the values of the relevant variables will be at time t=0) and to define and initialize variables (often vectors) that will hold all of the information we eventually might want to plot or use for other purposes. In our case, we are certainly going to plot voltage vs. time so let's define a time vector running from t=0 to t=t_end in time steps of size dt; and a corresponding voltage vector that will hold the voltage at each of these times. As a placeholder, let's initially assign the voltage vector to be all zeros. Note below that I put "_vect" on the end of the names of all vector variables – this notation helps to keep track of which variables are simply numbers (scalars) versus vectors.

```
%DEFINE INITIAL VALUES AND VECTORS TO HOLD RESULTS
t_vect = 0:dt:t_end;        %will hold vector of times
V_vect = zeros(1,length(t_vect));           %initialize the voltage vector
                                            %initializing vectors makes your code
                                            %run faster!
```

Aside: this initial setting up of the voltage vector to be the correct size is not strictly necessary but makes your code run faster. This is because it takes MATLAB a long time to create new vectors or change the size of old vectors (for the computer science whizzes, this is because creating or changing the size of vectors requires MATLAB to ask the computer for memory in which to store the vector, which is a slow and complicated process).

One more thing to add to the above section: We said that we initially want V = E_L so let's set the first element of V to this value (recall that the first element of the t_vector is t=0). It will be useful to have a variable corresponding to the index of the array so let's also define the variable i as the current element of V being assigned:

```
i = 1;   % index denoting which element of V is being assigned
V_vect(i)=E_L;  %first element of V, i.e. value of V at t=0
```

Good! Now we're ready to integrate equation (1). First, let's define the current injected at all times as I_e_vect. For now, set I_e_vect=zeros for all time. We'll try a few more interesting values soon. (Note: you may wonder why we didn't define I_e_vect in the parameters section. We could and maybe even should have, but here I am anticipating that we will later do a loop over various I_e_vect values. Stay tuned…).

```
%INTEGRATE THE EQUATION tau*dV/dt = -V + E_L + I_e*R_m
I_e_vect = zeros(1,length(t_vect));  %injected current [nA]
```

To do this integration, we now use the rule described in equation (3). We're clearly going to need to iterate this rule many times. That should be a clear signal to us that it's time to use a `for` loop that iterates over the values of t. Let's set that up and then fill in the inside of the loop later:

```
for t=dt:dt:t_end  % loop through values of t in steps of dt ms


end
```

Note that we start the loop at time dt because we already have the initial values t=0 and V(t=0)=E_L defined. Now let's fill in the inside of the loop. We need to first denote which element of V_vect is being updated (this should be one more than the last time through the loop, i.e. set i = i + 1) and then run our update rule to assign the appropriate value of V to this element of V_vect. The loop should read:

```
for t=dt:dt:t_end   %loop through values of t in steps of dt ms
   V_inf = E_L + I_e_vect(i)*R_m;  %value that V_vect is exponentially
                                   %decaying towards at this time step
   %next line does the integration update rule
   V_vect(i+1) = V_inf + (V_vect(i) - V_inf)*exp(-dt/tau);
   i = i + 1;  %add 1 to index, corresponding to moving forward 1 time step
```

```
end
```

Great! Now that we've assigned v, we're ready to plot. Add some plotting code next:

```
%MAKE PLOTS
figure(1)
plot(t_vect, V_vect);
title('Voltage vs. time');
xlabel('Time in ms');
ylabel('Voltage in mV');
```

Now go to your MATLAB command window and run your file. You should see a solid trace at -70 mV (if you don't, peek ahead and the code you should have typed will be summarized). This is exactly right: you assigned the voltage to start at rest and then didn't inject any current so the voltage stayed at rest. If you like, try playing with changing your initial voltage and see what happens (please remember to set it back to E_L before continuing on!).

Next, we said that we wanted to start stimulating at time t_StimStart = 100 ms and end at time t_StimEnd = 400 ms. Between 100 and 400 ms, let's set the elements of I_e_vect = I_StimPulse where I_StimPulse is the amplitude of the injected current during the stimulation. Let's set this to a value I_StimPulse = 1 nA. We could do this in 2 ways:

1) Within the loop use an if statement that says: if (t<100 || t>400) then assign the elements of I_e_vect = 0; else assign I_e_vect = I_StimPulse. (*Note:* || is MATLAB's symbol for the logical word OR; MATLAB's symbol for the logic word AND is &&). This is the most conceptually straightforward way but is not particularly efficient.

2) The more efficient way of doing the assignment is just to replace the line `I_e_vect = zeros(1,length(t_vect));` by an appropriate line defining the vector. Since dt=0.1 ms, we really want the first 1000 elements (from t=0.0 to t=99.9 ms) to equal zero; the next 3001 elements (from t=100.0 to t=400.0 ms) to equal I_StimPulse; and the final 1000 elements (from t=400.1 to t=500 ms) to equal zero.

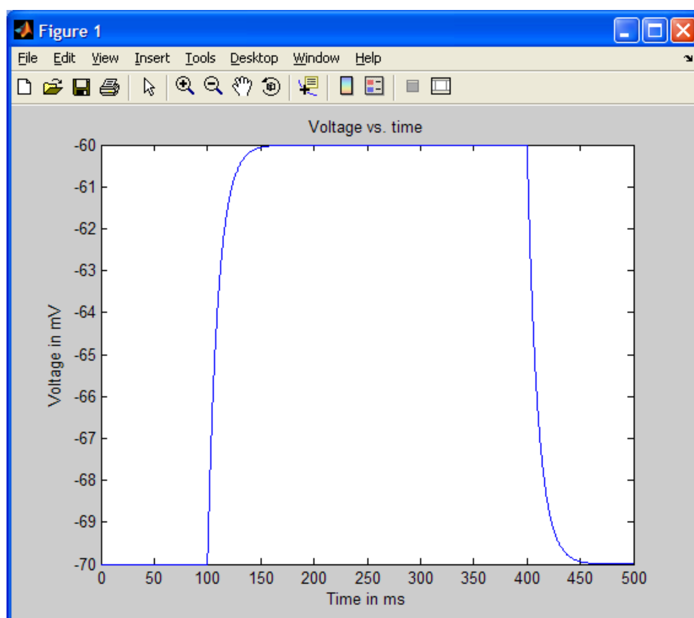
We can do the latter by the following lines (*replace* the previous `I_e_vect` line by this, and see next paragraph for detailed explanation):

```
I_Stim = 1;  %magnitude of pulse of injected current [nA]
I_e_vect = zeros(1,t_StimStart/dt);  %portion of I_e_vect from t=0 to t=t_StimStart
I_e_vect = [I_e_vect  I_Stim*ones(1,1+((t_StimEnd-t_StimStart)/dt))]; %add portion from
                                                        % t=t_StimStart to t=t_StimEnd
I_e_vect = [I_e_vect  zeros(1,(t_end-t_StimEnd)/dt)]; %add portion from
                                                    % t=t_StimEnd to t=t_end
```

In the second line, we set up the first portion of the vector as a row of zeros with the number of elements equal to t_StimStart/dt, which is the number of time points between zero and the stimulus start time. In the next line, we append to this vector a row of values I_Stim for all time point between t_StimStart and t_StimEnd (including the points t=tStimStart and the point t=tStimEnd. The "1+" covers this. For example, if tStimStart = 10, tStimEnd = 11, and dt = 0.1, then there would be eleven 5's appended to I_e_vector here.). In the final line above, we append to I_e_vect another row of zeros corresponding to times from t=tStimEnd to t=t_end. We could

actually have done this all in one line but it would have made our code harder to read without a major savings in efficiency. If you want to see the vector output from any of these lines, just remove the semicolon and see the print out (*Warning*: there are a lot of entries here! Use **Control-C**, which makes MATLAB stop whatever it is doing and return to the Command Prompt, to stop the output if you get tired of it scrolling across your screen). You could also type a length() statement to just check the lengths of these vectors. Checking lengths of vectors and size's of arrays is a very useful tool in debugging your code.

Now run your code. It should rise exponentially towards -60mV starting at t = 100, then decay back down exponentially at t = 400 (both rise and decay with time constants τ = 10 ms) as shown below. Try out some other values of I_Stim on your own to get a feeling for how big a voltage change you get for different values of I_Stim.



To summarize, your code at this stage should read:

```
% Lab 2: Build an integrate-and-fire model neuron and observe its spiking
%           for various levels of injected current

clear all;  %clear all variables
close all;  %close all open figures

%DEFINE PARAMETERS
dt = 0.1; %time step [ms]
t_end = 500; %total time of run [ms]
t_StimStart = 100;  %time to start injecting current [ms]
t_StimEnd = 400; %time to end injecting current [ms]
E_L = -70; %resting membrane potential [mV]
V_th = -55;  %spike threshold [mV]
V_reset = -75; %value to reset voltage to after a spike [mV]
V_spike = 20; %value to draw a spike to, when cell spikes [mV]
R_m = 10; %membrane resistance [MOhm]
```

```
tau = 10; %membrane time constant [ms]

%DEFINE INITIAL VALUES AND VECTORS TO HOLD RESULTS
t_vect = 0:dt:t_end;  %will hold vector of times
V_vect = zeros(1,length(t_vect));   %initialize the voltage vector
                                    %initializing vectors makes your code
                                    %run faster!
i = 1;   % index denoting which element of V is being assigned
V_vect(i)=E_L;  %first element of V, i.e. value of V at t=0

%INTEGRATE THE EQUATION tau*dV/dt = -V + E_L + I_e*R_m
I_Stim = 1;  %magnitude of pulse of injected current [nA]
I_e_vect = zeros(1,t_StimStart/dt);  %portion of I_e_vect from t=0 to t=t_StimStart
I_e_vect = [I_e_vect I_Stim*ones(1,1+((t_StimEnd-t_StimStart)/dt))]; %add portion from
                                                    % t=t_StimStart to t=t_StimEnd
I_e_vect = [I_e_vect zeros(1,(t_end-t_StimEnd)/dt)]; %add portion from
                                            % t=t_StimEnd to t=t_end
for t=dt:dt:t_end   %loop through values of t in steps of dt ms
   V_inf = E_L + I_e_vect(i)*R_m;  %value that V_vect is exponentially
                                  %decaying towards at this time step
   %next line does the integration update rule
   V_vect(i+1) = V_inf + (V_vect(i) - V_inf)*exp(-dt/tau);
   i = i + 1;  %add 1 to index, corresponding to moving forward 1 time step
end

%MAKE PLOTS
figure(1)
plot(t_vect, V_vect);
title('Voltage vs. time');
xlabel('Time in ms');
ylabel('Voltage in mV');
```

IV. Step 2: Add the spiking to the model and calculate the firing rate

Hopefully you noticed that, no matter how large you made I_Stim, your neuron did not spike. Next, we will add the code to make the neuron spike and then reset each time its voltage reaches the threshold value V_th. To do this, we need an if statement to detect when the voltage reaches V_th, and if so, we need to then reset the voltage back to V_reset. This can be done by adding the following immediately after the assignment V_vect(i+1) = V_inf + (V_vect(i) - V_inf)*exp(-dt/tau);

```
%if statement below says what to do if voltage crosses threshold
if (V_vect(i+1) > V_th)  %cell spiked
      V_vect(i+1) = V_reset;    %set voltage back to V_reset
end
```

Try running this code with I_Stim = 1.55. You should see the neuron reset its voltage each time it reaches V_th = -55 mV. This should occur 8 times. However, you are probably wondering, "Where are the beautiful spikes going up to some high voltage?" Well, in truth, the integrate-and-fire model never really assigns a voltage above V_th. Every time threshold is reached, it immediately resets the voltage to V_reset (which is our signal that a spike occurred at this time, if we were trying to count the number of spikes).

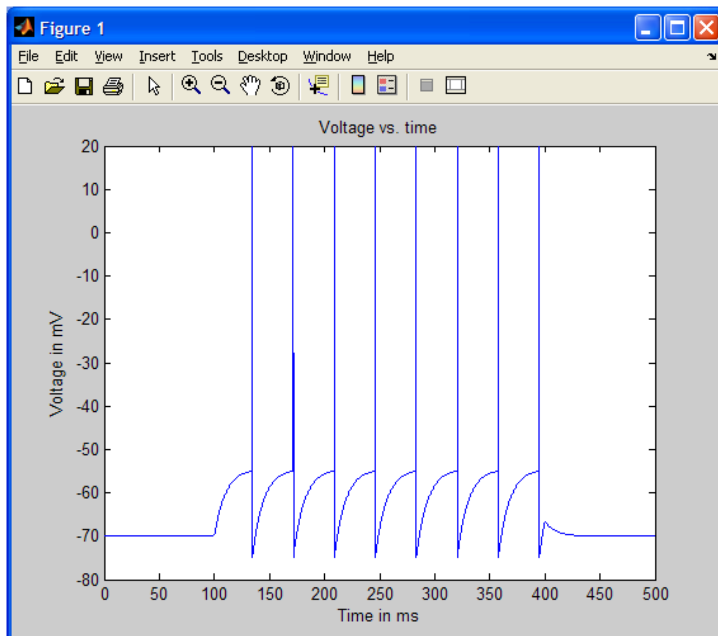
Well, we certainly want to make our plots prettier than that so let's "by hand" (well, aided by the computer…) assign a new vector we'll call V_plot_vect which replaces the first time point after threshold by a beautiful point at a voltage V=V_spike = 20 mV (or whatever value you find to be aesthetically appealing). We can do this by creating this vector in the "Define initial values…" section and also there assigning its first value to be equal to the value of the Voltage vector. Do this by modifying that section of the code to read:

```
%DEFINE INITIAL VALUES AND VECTORS TO HOLD RESULTS
t_vect = 0:dt:t_end; %will hold vector of times
V_vect = zeros(1,length(t_vect)); %initialize the voltage vector
                                   %initializing vectors makes your code run faster!
V_plot_vect = zeros(1,length(t_vect)); %pretty version of V_vect to be plotted, that displays a spike
                                        % whenever voltage reaches threshold
i = 1;   % index denoting which element of V is being assigned
V_vect(i)= E_L;  %first element of V, i.e. value of V at t=0
V_plot_vect(i) = V_vect(i);  %if no spike, then just plot the actual voltage V
```

Then modify the integration loop to assign this vector by replacing the if statement above by:

```
if (V_vect(i+1) > V_th)  %cell spiked
    V_vect(i+1) = V_reset;    %set voltage back to V_reset
    V_plot_vect(i+1) = V_spike; %set vector that will be plotted to show a spike here
else  %voltage didn't cross threshold so cell does not spike
    V_plot_vect(i+1) = V_vect(i+1); %plot the actual voltage
end
```

Now also change your plotting command to plot V_plot_vect rather than V_vect and run your program. You should see 8 beautiful spikes(!) like the following:



Finally, we would like to compute the *average firing rate* of the cell during the time of stimulation. A cell's average firing rate over a specified period of time is the number of spikes

produced over the specified time period: r_{ave}=(# of spikes)/(time period). A special situation is when we choose the period of time to be from immediately after one spike's occurrence to immediately after the next spike's occurrence. This time period between spikes is known as the *interspike interval* and is denoted by t_{isi}. The corresponding firing rate is $r_{isi} = 1/t_{isi}$, and this is what we calculated exactly in class for the integrate-and-fire model neuron receiving a constant stimulating current. Here, we will more simply calculate r_{ave} by counting the number of spikes that occurred during the stimulation period and then dividing by this time period. In the next section, we compare this value to the value of r_{isi} that we calculated in class.

To count the number of spikes, we add a new variable to our code called NumSpikes that we set initially to zero (since no spikes have occurred at the beginning of the simulation) and that we increase in value by 1 every time a spike occurs. We then divide this number by the duration of stimulation to get the firing rate in # spikes/ms. To convert from # spikes/ms to # spikes/sec we then multiply by 1000.

To do this, add just before your for loop the line:

```
NumSpikes = 0 %holds number of spikes that have occurred
```

Then add the following code within the if statement that identifies a spike:

```
NumSpikes = NumSpikes + 1 %add 1 to the total spike count
```

Finally, just after the end of your for loop add the line defining the average firing rate:

```
AveRate = 1000*NumSpikes/(t_StimEnd - t_StimStart) %gives average firing rate in [#/sec = Hz]
```

Leave off the semicolons so that the values output to your screen. Try this for a few values of I_Stim (to be realistic try to keep the firing rate between 0 and 100 Hz). For IStim=1.55 you should get a rate of 26.6667 Hz. [After trying this, add semicolons after the first 2 lines you assigned above so that NumSpikes doesn't keep printing to your screen.]

Your code for this section should now read:

```
% Lab 2: Build an integrate-and-fire model neuron and observe its spiking
%          for various levels of injected current

clear all;  %clear all variables
close all;  %close all open figures

%DEFINE PARAMETERS
dt = 0.1; %time step [ms]
t_end = 500; %total time of run [ms]
t_StimStart = 100;  %time to start injecting current [ms]
t_StimEnd = 400; %time to end injecting current [ms]
E_L = -70; %resting membrane potential [mV]
V_th = -55;  %spike threshold [mV]
V_reset = -75; %value to reset voltage to after a spike [mV]
V_spike = 20; %value to draw a spike to, when cell spikes [mV]
R_m = 10; %membrane resistance [MOhm]
tau = 10; %membrane time constant [ms]

%DEFINE INITIAL VALUES AND VECTORS TO HOLD RESULTS
```

```
t_vect = 0:dt:t_end; %will hold vector of times
V_vect = zeros(1,length(t_vect)); %initialize the voltage vector
                                  %initializing vectors makes your code run faster!
V_plot_vect = zeros(1,length(t_vect)); %pretty version of V_vect to be plotted, that displays a spike
                  % whenever voltage reaches threshold
i = 1;   % index denoting which element of V is being assigned
V_vect(i)= E_L;  %first element of V, i.e. value of V at t=0
V_plot_vect(i) = V_vect(i);  %if no spike, then just plot the actual voltage V

%INTEGRATE THE EQUATION tau*dV/dt = -V + E_L + I_e*R_m
I_Stim = 1.55;  %magnitude of pulse of injected current [nA]
I_e_vect = zeros(1,t_StimStart/dt);  %portion of I_e_vect from t=0 to t=t_StimStart
I_e_vect = [I_e_vect I_Stim*ones(1,1+((t_StimEnd-t_StimStart)/dt))]; %add portion from
                                                   % t=t_StimStart to t=t_StimEnd
I_e_vect = [I_e_vect zeros(1,(t_end-t_StimEnd)/dt)]; %add portion from
                                              % t=t_StimEnd to t=t_end
NumSpikes = 0; %holds number of spikes that have occurred
for t=dt:dt:t_end   %loop through values of t in steps of dt ms
   V_inf = E_L + I_e_vect(i)*R_m;  %value that V_vect is exponentially
                                   %decaying towards at this time step
   %next line does the integration update rule
   V_vect(i+1) = V_inf + (V_vect(i) - V_inf)*exp(-dt/tau);
   %if statement below says what to do if voltage crosses threshold
   if (V_vect(i+1) > V_th)  %cell spiked
      V_vect(i+1) = V_reset;    %set voltage back to V_reset
      V_plot_vect(i+1) = V_spike; %set vector that will be plotted to show a spike here
      NumSpikes = NumSpikes + 1; %add 1 to the total spike count
   else   %voltage didn't cross threshold so cell does not spike
      V_plot_vect(i+1) = V_vect(i+1); %plot the actual voltage
   end
   i = i + 1;  %add 1 to index, corresponding to moving forward 1 time step
end
AveRate = 1000*NumSpikes/(t_StimEnd - t_StimStart) %gives average firing rate in [#/sec = Hz]

%MAKE PLOTS
figure(1)
plot(t_vect, V_plot_vect);
title('Voltage vs. time');
xlabel('Time in ms');
ylabel('Voltage in mV');
```

V. Step 3: Compare $r_{isi,theory}$ to r_{ave}

Save your work from the last section and then use Save As… to rename the file you are working on to something new (e.g. to IntAndFire3.m).

Next we'd like to compare the theoretical value for the firing rate of the integrate-and-fire neuron $r_{isi}=1/t_{isi}$ (equation (5)) to the value of r_{ave} we calculated above. We'll do this for several values of I_Stim (i.e. of I_e in equation (5)). How are we going to efficiently run our code for several different values of I_Stim? You guessed it…use another for loop!

First, take a moment to indent all the lines below (but not including) I_Stim = 1.55 by selecting them and then clicking Text >> Increase Indent. This will make the following code more readable.

Now let's turn our code into a for loop by defining a vector of stimuli and then looping over it. Erase the I_Stim = 1.55 line and replace it by:

```
I_Stim_vect = 1.43:0.04:1.63;  %magnitudes of pulse of injected current [nA]
for I_Stim = I_Stim_vect;  %loop over different I_Stim values
```

Also add as the last line of your entire code:

```
end   %for I_Stim
```

This is needed to finish the for loop. The code you just added will allow you to loop over 6 values of I_Stim from 1.43 to 1.63. Each time we loop we're going to want to re-initialize the voltage vector and voltage plotting vector so cut and paste your previously typed lines:

```
i = 1;   % index denoting which element of V is being assigned
V_vect(i)= E_L;  %first element of V, i.e. value of V at t=0
V_plot_vect(i) = V_vect(i);  %if no spike, then just plot the actual voltage V
```

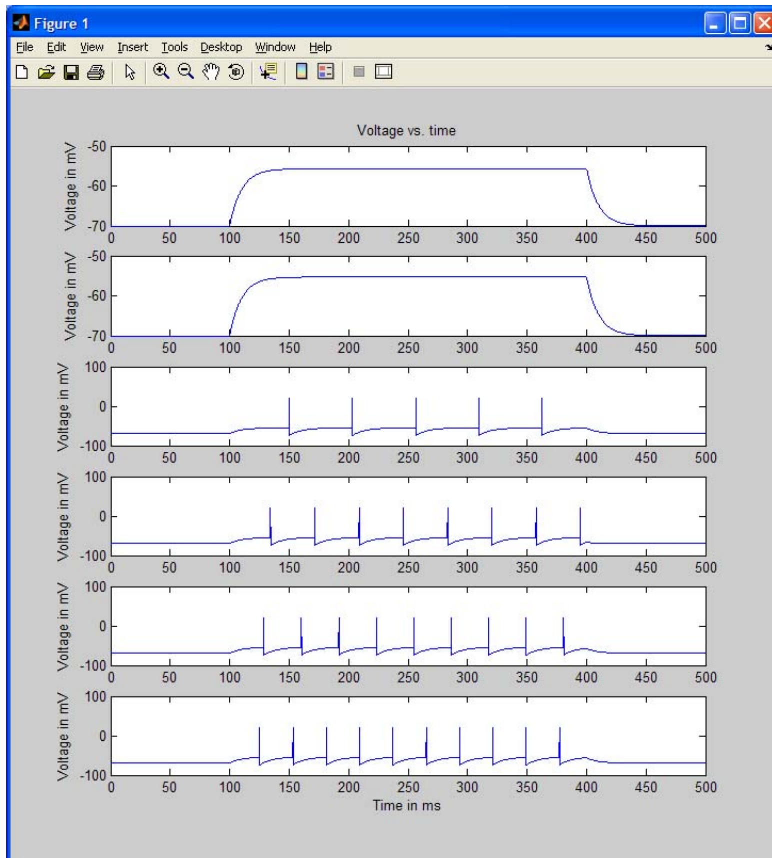
to be the first lines of your new for loop. Now we're also going to want to make separate plots for each run so let's define a variable PlotNum corresponding to the number of plots. Initialize PlotNum to zero above the for loop, and then have it increase by 1 every time we step through the for loop. Your %INTEGRATE THE EQUATION code should now start with:

```
PlotNum=0;
I_Stim_vect = 1.43:0.04:1.63;
for I_Stim = I_Stim_vect;  %magnitude of pulse of injected current [nA]
  PlotNum = PlotNum + 1;
  i = 1;   % index denoting which element of V is being assigned
  V_vect(i)= E_L;  %first element of V, i.e. value of V at t=0
  V_plot_vect(i) = V_vect(i);  %if no spike, then just plot the actual voltage V
```

Now let's set up to make an array of plots: Below the command figure(1), make your code now read:

```
  subplot(length(I_Stim_vect),1,PlotNum)
  plot(t_vect, V_plot_vect);
  if (PlotNum == 1)
    title('Voltage vs. time');
  end
  if (PlotNum == length(I_Stim_vect))
    xlabel('Time in ms');
  end
  ylabel('Voltage in mV');
end  %for I_Stim
```

The subplot command sets up an array of plots with length(I_Stim_vect) rows and 1 column. The if statements make the title only plot above the first subplot and the x-axis label only plot below the last plot. Try running your code now. This should produce the following panels, the first 2 with no spikes and the latter ones with increasing numbers of spikes:



This should look like the effect of increasing light intensity on the spiking of neurons of the eye in Hartline's paper! In this context, we are assuming that the effect of increasing light is simply to increase the current injected into the neuron and thereby to increase its firing rate.

Now let's add another figure that plots the theoretical firing rate vs. I_e curve for values of I_e above firing rate threshold I_threshold = (V_th - E_L)/R_m. This is easily done by defining a vector of injected currents (let's call it I_vect_long since it will contain many points) and then typing in the ugly formula for r_{isi} from equation (5)). If we want to plot from just above I_threshold to I_e = 1.8 in fine steps of 0.001, the code is (add to end of your code):

```
%COMPARE R_AVE TO R_ISI
I_threshold = (V_th - E_L)/R_m; %current below which cell does not fire
I_vect_long = (I_threshold+0.001):0.001:1.8; %vector of injected current for producing theory plot
r_isi = 1000./(tau*log((V_reset - E_L - I_vect_long*R_m)./(V_th - E_L - I_vect_long*R_m)));
figure(2)
plot(I_vect_long,r_isi)
title('Comparison of r_{isi} vs. I_e and r_{ave} vs. I_e')
xlabel('Injected current (nA)')
ylabel('r_{isi} or r_{ave} (Hz)')
```

The formula is quite ugly (but correct…). Things to note here are: 1) that we need to use ./ to tell MATLAB that we are dividing all the values element by element in producing r_{isi} from I_vect_long, and 2) That the underscore in figure labels tells MATLAB to subscript and the braces tell MATLAB to subscript everything included in the braces (try leaving them off and see what happens). Try running your code now.

Ok—final step (hooray!). We now want to add onto this graph the values of r_ave corresponding to the runs shown in figure 1. To do this we should turn r_ave into a vector with elements indexed by PlotNum (i.e. the first element of r_ave will correspond to the first plot, the second element to the 2nd plot, and so on). Thus replace the AveRate assignment line by:

```
AveRate_vect(PlotNum) = 1000*NumSpikes/(t_StimEnd - t_StimStart) %gives average firing
                                                                  %rate in [#/sec = Hz]
```

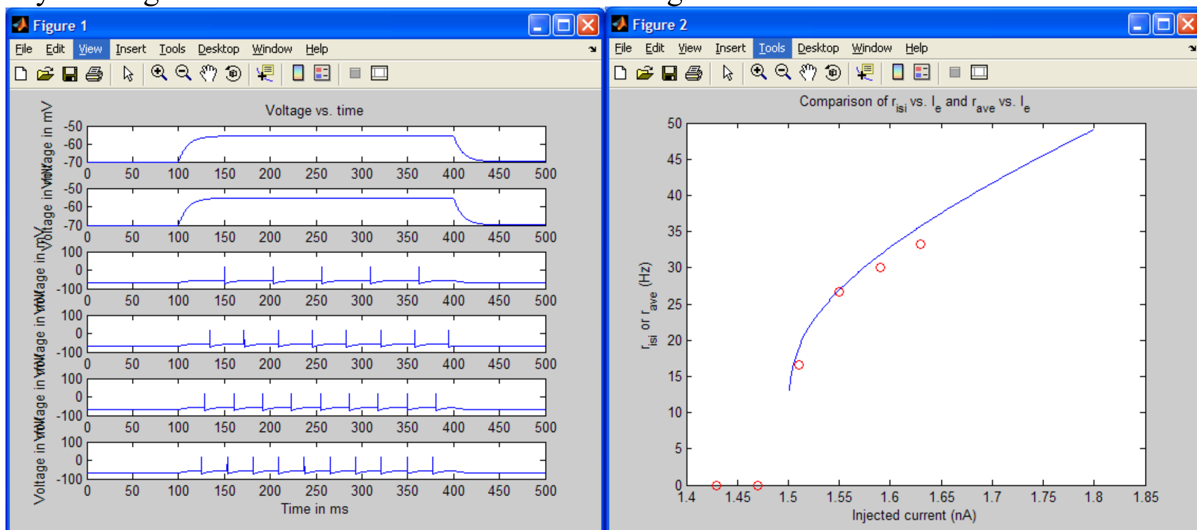
Finally, let's add a plot of the AveRate vs. I_Stim points (let's do red circles) to the theory plot by telling this graph to hold on. I.e., following the first plot command line of figure 2, add the 2 lines:

```
hold on
plot(I_Stim_vect,AveRate_vect,'ro')
```

Finally, make the very last line of the .m file :

```
hold off  %to ensure that doesn't keep this data the next time you want to plot something
```

Try running this code. You should see the following:



How well does r_{ave} match r_{isi}? If we started our time window of counting spikes for computing r_{ave} when the voltage was at V_{reset} (e.g. just after a spike) and ended our time window just after another spike, then these two measures should be exactly the same. However, if our time window of counting is not exactly equal to a constant number of interspike intervals, then there may be a difference. For example, if we ended our time window just before a spike, then for a

300 ms window, this will lower r_{ave} by (1 spike)/(300 ms) = 1/(0.3 sec) = 3.33 Hz. r_{ave} could also be higher than r_{isi} if, for example, we were to start our time window for counting spikes at a time when the voltage is higher than V_{reset} and end our time window just after a spike (e.g. if our time window went from just before a spike to just after a spike, we could count 2 spikes in an interval just barely longer than t_{isi}). Looking at the 3rd, 5th, and 6th panels of Figure 1, the cell was about to spike when we cut off the stimulus, so our r_{ave} rate is smaller than $r_{isi}=1/t_{isi}$. Try running for a longer time and see if this helps out.

VI. Code summary

Your final code from this lab should be:

```
% Lab 2: Build an integrate-and-fire model neuron and observe its spiking
%           for various levels of injected current

clear all;  %clear all variables
close all;  %close all open figures

%DEFINE PARAMETERS
dt = 0.1; %time step [ms]
t_end = 500; %total time of run [ms]
t_StimStart = 100;  %time to start injecting current [ms]
t_StimEnd = 400; %time to end injecting current [ms]
E_L = -70; %resting membrane potential [mV]
V_th = -55;  %spike threshold [mV]
V_reset = -75; %value to reset voltage to after a spike [mV]
V_spike = 20; %value to draw a spike to, when cell spikes [mV]
R_m = 10; %membrane resistance [MOhm]
tau = 10; %membrane time constant [ms]

%DEFINE INITIAL VALUES AND VECTORS TO HOLD RESULTS
t_vect = 0:dt:t_end; %will hold vector of times
V_vect = zeros(1,length(t_vect)); %initialize the voltage vector
                              %initializing vectors makes your code run faster!
V_plot_vect = zeros(1,length(t_vect)); %pretty version of V_vect to be plotted, that displays a spike
                    % whenever voltage reaches threshold

%INTEGRATE THE EQUATION tau*dV/dt = -V + E_L + I_e*R_m
PlotNum=0;
I_Stim_vect = 1.43:0.04:1.63; %magnitudes of pulse of injected current [nA]
for I_Stim = I_Stim_vect;  %loop over different I_Stim values
   PlotNum = PlotNum + 1;
   i = 1;   % index denoting which element of V is being assigned
   V_vect(i)= E_L;  %first element of V, i.e. value of V at t=0
   V_plot_vect(i) = V_vect(i);  %if no spike, then just plot the actual voltage V
   I_e_vect = zeros(1,t_StimStart/dt);  %portion of I_e_vect from t=0 to t=t_StimStart
   I_e_vect = [I_e_vect I_Stim*ones(1,1+((t_StimEnd-t_StimStart)/dt))]; %add portion from
                                                      % t=t_StimStart to t=t_StimEnd
   I_e_vect = [I_e_vect zeros(1,(t_end-t_StimEnd)/dt)]; %add portion from
                                          % t=t_StimEnd to t=t_end
   NumSpikes = 0; %holds number of spikes that have occurred
   for t=dt:dt:t_end   %loop through values of t in steps of dt ms
      V_inf = E_L + I_e_vect(i)*R_m;  %value that V_vect is exponentially
                                 %decaying towards at this time step
```

```
        %next line does the integration update rule
        V_vect(i+1) = V_inf + (V_vect(i) - V_inf)*exp(-dt/tau);
        %if statement below says what to do if voltage crosses threshold
        if (V_vect(i+1) > V_th)  %cell spiked
            V_vect(i+1) = V_reset;    %set voltage back to V_reset
            V_plot_vect(i+1) = V_spike; %set vector that will be plotted to show a spike here
            NumSpikes = NumSpikes + 1; %add 1 to the total spike count
        else   %voltage didn't cross threshold so cell does not spike
            V_plot_vect(i+1) = V_vect(i+1); %plot the actual voltage
        end
        i = i + 1;  %add 1 to index, corresponding to moving forward 1 time step
    end
    AveRate_vect(PlotNum) = 1000*NumSpikes/(t_StimEnd - t_StimStart) %gives average firing
                                                                       %rate in [#/sec = Hz]

    %MAKE PLOTS
    figure(1)
    subplot(length(I_Stim_vect),1,PlotNum)
    plot(t_vect, V_plot_vect);
    if (PlotNum == 1)
        title('Voltage vs. time');
    end
    if (PlotNum == length(I_Stim_vect))
        xlabel('Time in ms');
    end
    ylabel('Voltage in mV');
end   %for I_Stim

%COMPARE R_AVE TO R_ISI
I_threshold = (V_th - E_L)/R_m; %current below which cell does not fire
I_vect_long = (I_threshold+0.001):0.001:1.8; %vector of injected current for producing theory plot
r_isi = 1000./(tau*log((V_reset - E_L - I_vect_long*R_m)./(V_th - E_L - I_vect_long*R_m)));
figure(2)
plot(I_vect_long,r_isi)
hold on
plot(I_Stim_vect,AveRate_vect,'ro')
title('Comparison of r_{isi} vs. I_e and r_{ave} vs. I_e')
xlabel('Injected current (nA)')
ylabel('r_{isi} or r_{ave} (Hz)')
hold off  %to ensure that doesn't keep this data the next time you want to plot something
```