```
% Tutorial 2.1
% Created by Jonathan Lindbloom, 2/7/2019
% Email: jlindbloom@smu.edu

clear all

% 1.a.i Define parameters.
global e_leak r_membrane c_membrane v_threshold v_reset tau_m
e_leak = -70e-3;
r_membrane = 5e6;
c_membrane = 2e-9;
v_threshold = -50e-3;
v_reset = -65e-3;
tau_m = c_membrane*r_membrane;  % Time constant.

% 1.a.ii Create time vector.
global dt
dt = 0.0001;
t = 0:dt:2;

% 1.a.iii Create vector for membrane potential.
v = zeros(1, length(t));

% 1.a.iv Set initial value of membrane potential to E_leak.
v(1) = e_leak;

% 1.a.v Create a vector for applied current I_app, of size identical
 to the
% time vector t, with each entry set to a constant value I_0.

I_th = (v_threshold - e_leak)/r_membrane;   % Threshold value computed
 in 1.b.

I_0 = I_th*1.01;
I_app = I_0*ones(1,length(t));

% 1.a.vi/vii/viii . Set up for loop to run the 4th-order Runge Kutta
% method, reseting the membrane potential to the reset value if
 threshold exceeded.

v = runsim(v, t, dt, I_app, 0); % See functions at end of script.

% 1.b. Find minimum applied current needed for the neuron to produce
 spikes (from eq 2.11).
I_th = (v_threshold - e_leak)/r_membrane;
minimum_threshold = sprintf('The minimum applied current needed for
 the neuron to produce spikes is %0.5e .', I_th);
disp(minimum_threshold)

% 1.b. Model with applied current slightly less than threshold.

v = zeros(1, length(t));    % Reset simulation.
```

```
v(1) = e_leak;
I_0 = I_th*0.99;
I_app = I_0*ones(1,length(t));
v = runsim(v, t, dt, I_app, 0);

f1 = figure;
figure(f1);
plot(t(1:301), v(1:301));
title(sprintf('Leaky Integrate-and-Fire Moodel with I_{{app}} =
 %0.5e', I_0));
xlabel('Time Elapsed (s)');
ylabel('Voltage (mV)');
saveas(f1, sprintf('1b_slightly_less_than_threshold_Iapp_%0.5e.png',
 I_0));

% 1.b. Model with applied current slightly greater than threshold.

v = zeros(1, length(t));    % Reset simulation.
v(1) = e_leak;
I_0 = I_th*1.1;
I_app = I_0*ones(1,length(t));
v = runsim(v, t, dt, I_app, 0);

f2 = figure;
figure(f2);
plot(t(1:301), v(1:301));
title(sprintf('Leaky Integrate-and-Fire Moodel with I_{{app}} =
 %0.5e', I_0));
xlabel('Time Elapsed (s)');
ylabel('Voltage (mV)');
saveas(f2, sprintf('1b_slightly_greater_than_threshold_Iapp_
%0.5e.png', I_0));

% 1.c. Run simulation for varying applied current and generate f-I
 curve.

currents = [I_th*1.05 I_th*1.1 I_th*1.15 I_th*1.2 I_th*1.25 I_th*1.3
 I_th*1.35 I_th*1.4 I_th*1.45 I_th*1.5];
rates = zeros(1, length(currents));
for n = 1:length(currents)
    % Reset simulation each time.
    v = zeros(1, length(t));
    v(1) = e_leak;
    I_0 = currents(n);
    I_app = I_0*ones(1,length(t));
    v = runsim(v, t, dt, I_app, 0);
    numspikes = fire_count(v, v_reset);
    rate = numspikes/2;
    rates(n) = rate;
end


% 1.d. Generate theoretical f-I curve data.
```

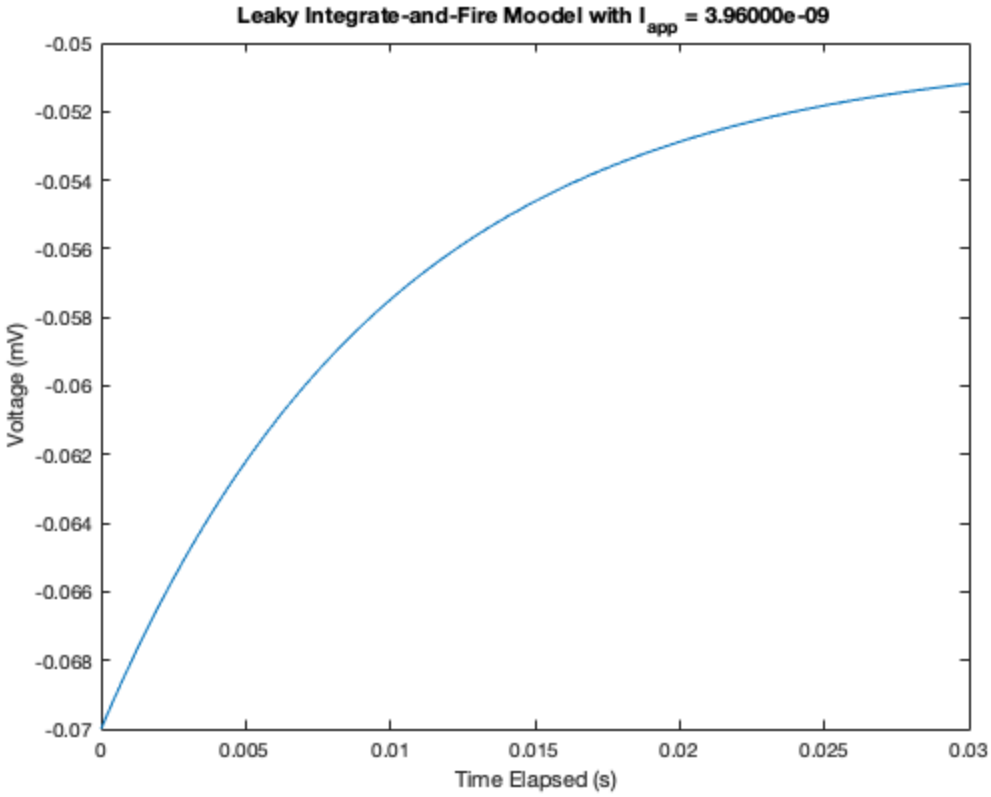
```
currents = [I_th*1.05 I_th*1.1 I_th*1.15 I_th*1.2 I_th*1.25 I_th*1.3
 I_th*1.35 I_th*1.4 I_th*1.45 I_th*1.5];
theoretical_rates = zeros(1, length(currents));
for n = 1:length(currents)
    theoretical_rates(n) = theoretical_rate(currents(n));
end


% Generate plot for 1.c. and 1.d.
f3 = figure;
figure(f3);
scatter(currents, rates);
hold on;
scatter(currents, theoretical_rates);
legend('Simulated', 'Theoretical');
title('Simulated vs. Theoretical f-I Curve');
xlabel('I_{app}');
ylabel('Firing Rate (Hz)');
saveas(f3, 'Simulated_vs_Theoretical_fI_Curve.png');

% 2.a. This step is included in the definition of the runsim function
 (see
% end of script).

% 2.b. Plot fI curve for two sigmas, comparing to fI curve with no
 noise.

sigma1 = 0.05;
rates_s1 = zeros(1, length(currents));
for n = 1:length(currents)
    % Reset simulation each time.
    v = zeros(1, length(t));
    v(1) = e_leak;
    I_0 = currents(n);
    I_app = I_0*ones(1,length(t));
    v = runsim(v, t, dt, I_app, sigma1);
    numspikes = fire_count(v, v_reset);
    rate = numspikes/2;
    rates_s1(n) = rate;
end

sigma2 = 0.1;
rates_s2 = zeros(1, length(currents));
for n = 1:length(currents)
    % Reset simulation each time.
    v = zeros(1, length(t));
    v(1) = e_leak;
    I_0 = currents(n);
    I_app = I_0*ones(1,length(t));
    v = runsim(v, t, dt, I_app, sigma2);
    numspikes = fire_count(v, v_reset);
    rate = numspikes/2;
    rates_s2(n) = rate;
end
```

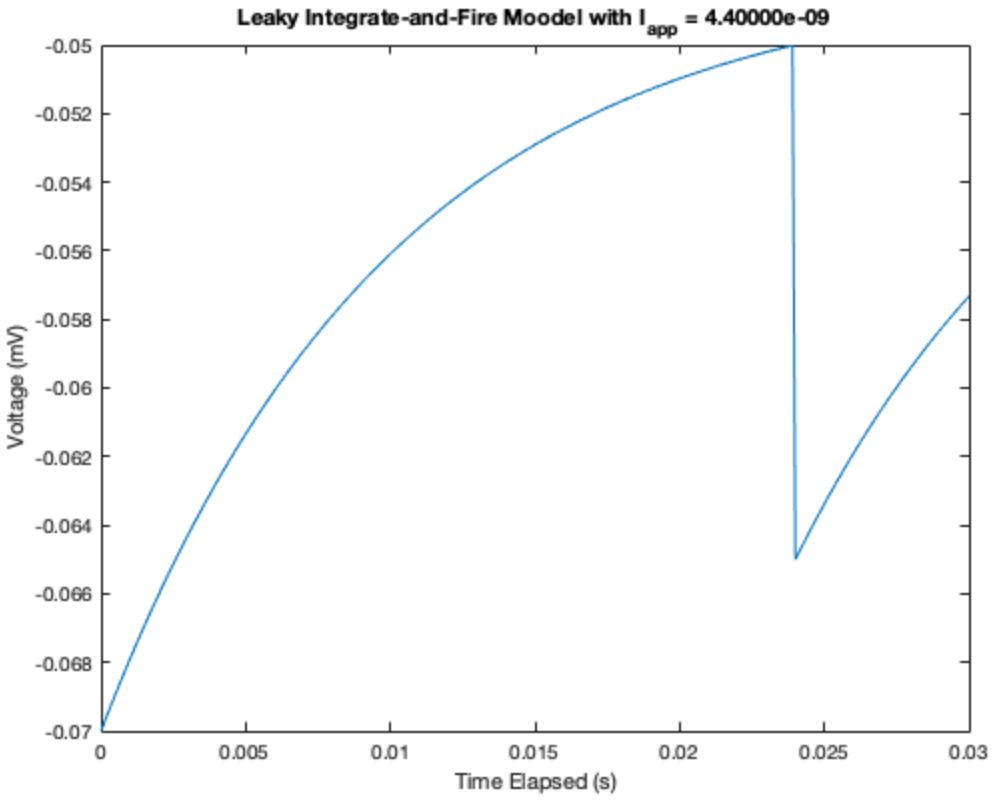
```
sigma3 = 0.13;
rates_s3 = zeros(1, length(currents));
for n = 1:length(currents)
    % Reset simulation each time.
    v = zeros(1, length(t));
    v(1) = e_leak;
    I_0 = currents(n);
    I_app = I_0*ones(1,length(t));
    v = runsim(v, t, dt, I_app, sigma3);
    numspikes = fire_count(v, v_reset);
    rate = numspikes/2;
    rates_s3(n) = rate;
end

sigma4 = 0.15;
rates_s4 = zeros(1, length(currents));
for n = 1:length(currents)
    % Reset simulation each time.
    v = zeros(1, length(t));
    v(1) = e_leak;
    I_0 = currents(n);
    I_app = I_0*ones(1,length(t));
    v = runsim(v, t, dt, I_app, sigma4);
    numspikes = fire_count(v, v_reset);
    rate = numspikes/2;
    rates_s4(n) = rate;
end

f4 = figure;
figure(f4);
scatter(currents, rates);
hold on;
scatter(currents, rates_s1);
hold on;
scatter(currents, rates_s2);
hold on;
scatter(currents, rates_s3);
hold on;
scatter(currents, rates_s4);
legend('No Noise', strcat('\sigma_1 = ', num2str(sigma1)),
 strcat('\sigma_2 = ', num2str(sigma2)), strcat('\sigma_3 = ',
 num2str(sigma3)), strcat('\sigma_4 = ', num2str(sigma4)));
title('Simulated f-I Curve with Noise');
xlabel('I_{app}');
ylabel('Firing Rate (Hz)');
saveas(f4, 'Simulated_fI_Curve_with_Noise.png');

% It apears that an increase in the parameter sigma, which is
 proportional
% to the standard deviation of the voltage noise, leads to an increase
 in
% the average firing rate that is approximately linear with the
 increase in
```

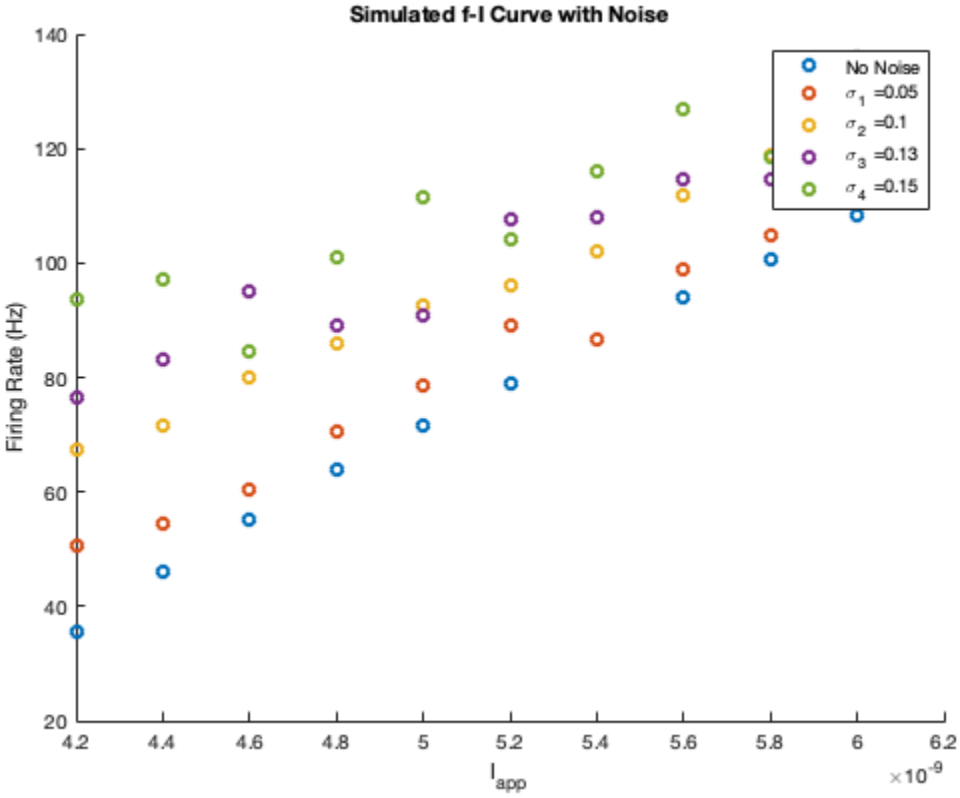
```
% sigma. However, after a point the increases in sigma appear to bend
 the
% shape of the no-noise f-I curve.


% Function Definitions:

function value = F(v, t, i_applied)
global e_leak r_membrane c_membrane
% Evaluates the RHS function of the ODE with given v, t, and I_app.
% Used for generating the RK4 solution.
value = ((e_leak - v)/(r_membrane*c_membrane)) + ((i_applied)/
c_membrane);
end


function v_simulated = runsim(v, t, t_delta, i_applied, sigma)
% Simulates the membrane potential with given inputs. If no noise
% fluctuations desired, pass 0 as the sigma argument.
global v_threshold v_reset dt
v_simulated = zeros(1, length(v));
v_simulated(1) = v(1);
temp = length(v)-1;
for n = 1:temp
    K0 = t_delta*(F(v_simulated(n), t(n), i_applied(n)));
    K1 = t_delta*(F(v_simulated(n) + K0/2, t(n) + dt/2,
 i_applied(n)));
    K2 = t_delta*(F(v_simulated(n) + K1/2, t(n) + dt/2,
 i_applied(n)));
    K3 = t_delta*(F(v_simulated(n) + K2, t(n) + dt, i_applied(n)));
    K4 = (K0 + 2*K1 + 2*K2 + K3)/6;
    with_noise = K4 + randn(1)*sigma*sqrt(dt);  % Noise term.

    if (v_simulated(n) + with_noise) >= v_threshold
        v_simulated(n+1) = v_reset;
    else
        v_simulated(n+1) = v_simulated(n) + with_noise;
    end
end
end

function count = fire_count(v, v_reset)
% Counts the number of spikes by counting the number of times
% the membrane potential takes on the reset value.
count = 0;
for n = 1:length(v)
    if v(n) == v_reset
        count = count + 1;
    end
end
end

function rate = theoretical_rate(current)
% Computes the theoretical firing rate.
```

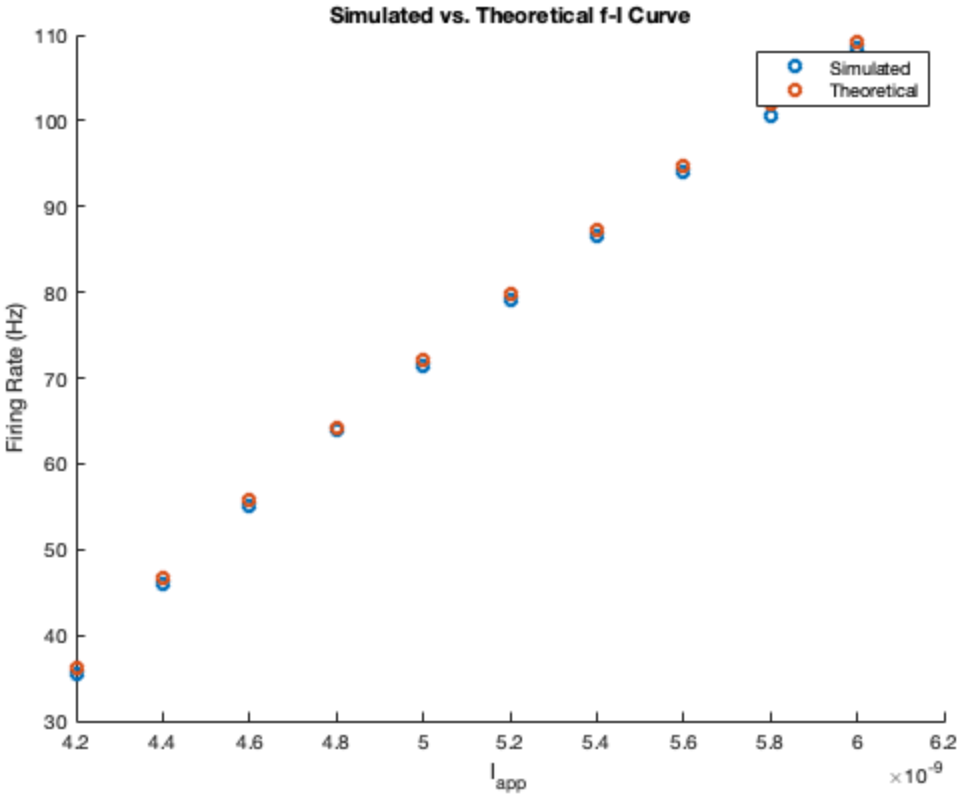
```
global r_membrane e_leak v_reset v_threshold tau_m

if ((current*r_membrane + e_leak - v_reset) > 0) &
 ((current*r_membrane + e_leak - v_threshold) > 0)
    rate = (tau_m*log(current*r_membrane + e_leak - v_reset) -
 tau_m*log(current*r_membrane + e_leak - v_threshold));
    rate = 1/rate;
end
end
```

The minimum applied current needed for the neuron to produce spikes is 4.00000e-09 .




Leaky Integrate-and-Fire Moodel with I_app = 4.40000e-09
Voltage (mV)
-0.05
-0.052
-0.054
-0.056
-0.058
-0.06
-0.062
-0.064
-0.066
-0.068
-0.07
0
0.005
0.01
0.015
0.02
0.025
0.03
Time Elapsed (s)


Simulated f-I Curve with Noise
Firing Rate (Hz)
140
120
100
80
60
40
20
4.2
4.4
4.6
4.8
5
5.2
5.4
5.6
5.8
6
6.2
I_app
×10^-9
No Noise
σ1 =0.05
σ2 =0.1
σ3 =0.13
σ4 =0.15



Published with MATLAB® R2018a