IN THE NAME OF GOD

HW1:

Neuroscience, Learning, Memory, Cognition Course

Sharif University of Technology

SaraRezanezhad99101643

March2023

PART I: PYTHON PRACTICE

1)Working with arrays using PyLab

- Pylab is a module that provides a Matlab like namespace by importing functions from the modules Numpy and Matplotlib. Numpy provides efficient numerical vector calculations based on underlying Fortran and C binary libraries. Matplotlib contains functions to create visualizations of data.

- There're two different ways to multiply two arrays (or variables, objecst or ...) ; In first way we can easily use operator * from pylab package or we can create an object from numpy and then use multiply() function.
As we can see there's no difference in the result (in both way element of a matrix is multiplied by the equivalent element of the second matrix)

```
from pylab import *
a = array([[1, 2, 3, 4]])
b = array( [[0, 4, 7, 6]] )
print(a*b)
```

```
[[ 0  8 21 24]]
```

```
import numpy as geek
a = array([[1, 2, 3, 4]])
b = array( [[0, 4, 7, 6]] )
ans = geek.multiply(a,b)
print(ans)
```

```
[[ 0  8 21 24]]
```

2) Defining a new function.

In this part of exercise we simply define a function and then by using symbol ? we can extract information about this specific function.

We set an example in the end.

```
def my_square_function(x, c):
    return   x*x + c
```

```
my_square_function?
```

```
Signature: my_square_function(x, c)
Docstring: <no docstring>
File:      /tmp/ipykernel_112/3890668996.py
Type:      function
```

```
my_square_function(5, 15)
```

```
40
```

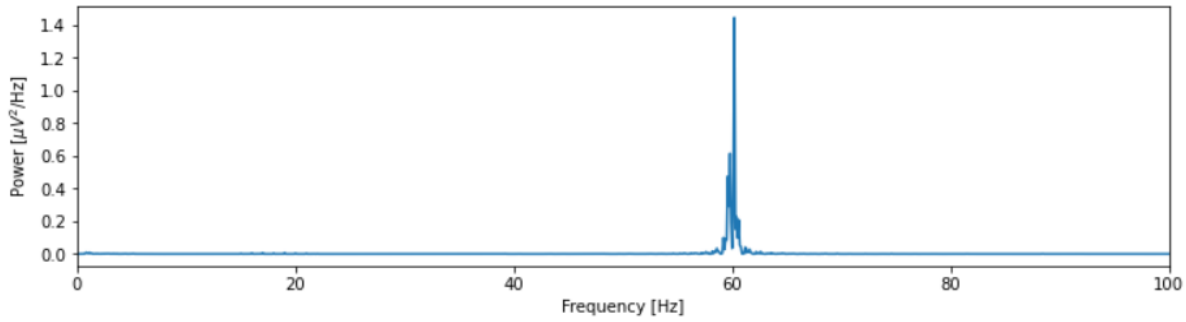
3)Loading matlab data (.mat) and plot power spectrum

The missing part of the code is supposed to performs spectral analysis of a signal x using the fast Fourier transform (FFT). The output is the signal's power spectral density, which gives the distribution of power across different frequency bands.

Step by step explanation:

- xf = fft(x-x.mean()): This line computes the FFT of the centered signal, i.e., x with its mean subtracted. The fft function computes the discrete Fourier transform of the signal, and returns an array of complex numbers representing the signal's frequency components.
- Sxx = 2 * dt 2 / T * (xf * xf.conj()): This line computes the power spectral density of the signal, which is proportional to the squared absolute value of the FFT. dt denotes the time step, and T denotes the total length of the signal. The factor 2 * dt 2 / T is a normalization term, which ensures that the PSD has units of power per frequency. The conj method returns the complex conjugate of xf, which is needed because the FFT output is generally complex.
- Sxx = Sxx[:int(len(x) / 2)]: This line takes only the first half of the PSD output, which corresponds to the positive-frequency components of the signal. The second half contains the negative-frequency components, which are redundant due to the symmetry of the FFT output for real-valued signals.

Overall, the code transforms the signal into the frequency domain and obtains its power spectral density, which can be used to analyze its frequency content and identify dominant frequencies or patterns.



I used an example code ,which define the PSD function, to help my understanding.(It was not used)

```
def PSD(f, signal, Fs):
    t = 1/Fs        # Sample spacing
    T = len(signal) # Signal duration

    s = np.sum([signal[i] * np.exp(-1j*2*np.pi*f*i*t) for i in range(T)])

    return t**2 / T * np.abs(s)**2
```

PART II: NEURON MODELS

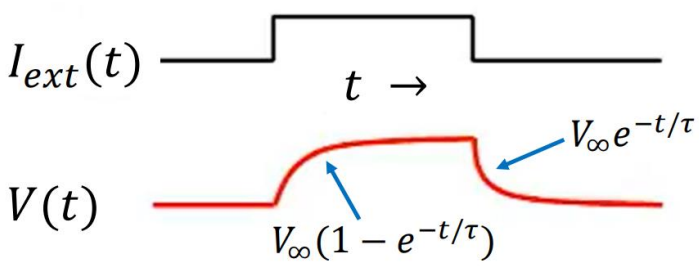
Coding Exercise 1: Python code to simulate the LIF neuron

By using $\tau_m \frac{dV}{dt} = -(V - E_L) + \frac{I}{g_L}$, and multiply dt/τ_m to the right part of equation we can fill the missing part of the code.

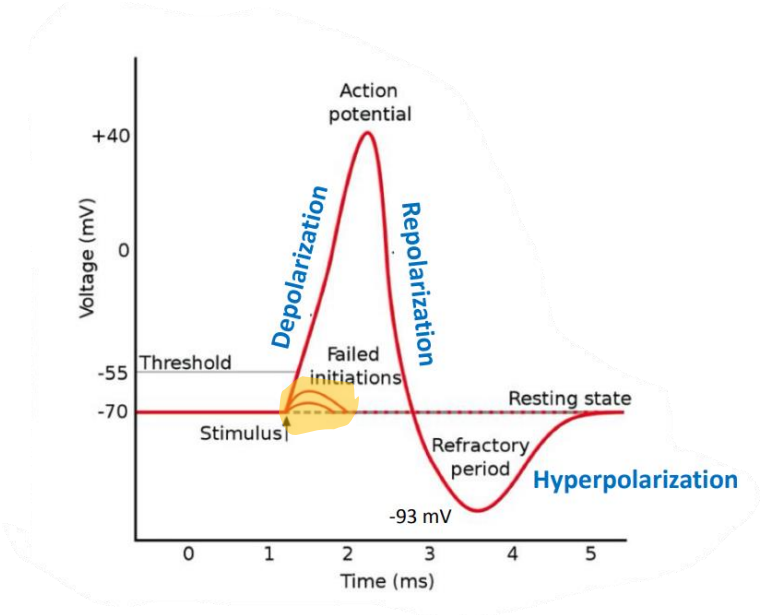
In the "Update the membrane potential" part, we use current voltage(v[it]) plus changing of voltage (dv) to obtain new amount of voltage.

As we know for pulse input current , there's a similar pulse output which doesn't reach to the threshold voltage(=-55v) because input current isn't enough.

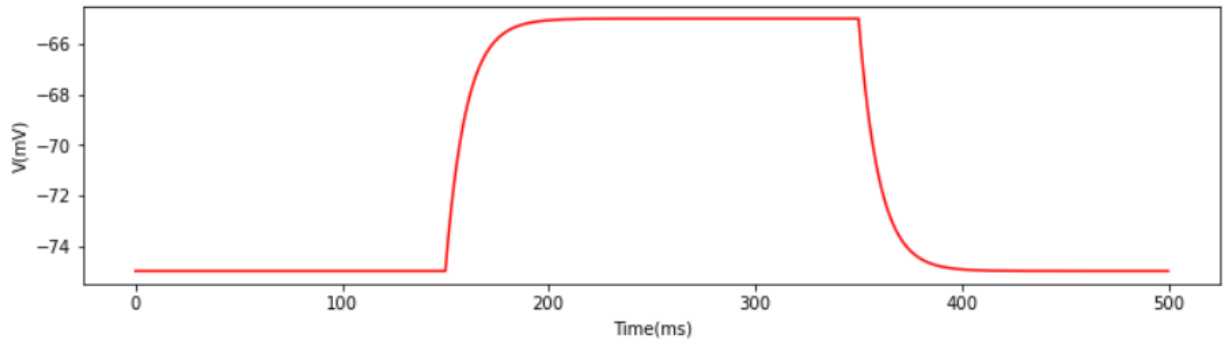
- For an input current in the form of a pulse:



So as we can see in the below plot ,our stimulus had faild intiation:



After introducing insufficient current into the system, since the voltage cannot reach the threshold, it returns to the resting state(=-75v).

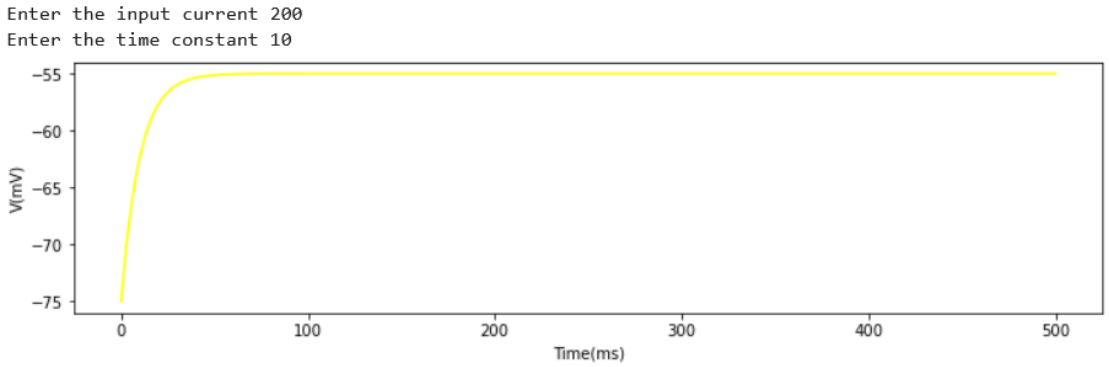


Response of an LIF model to different types of input currents

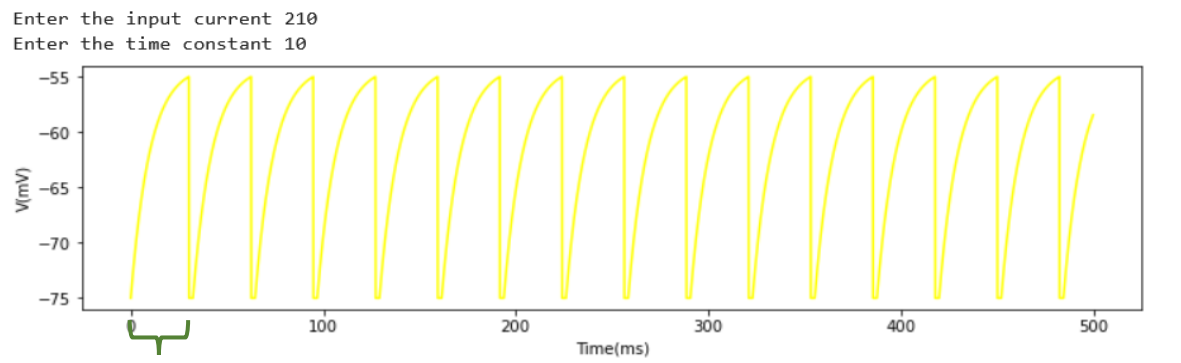
Parameter exploration of Direct current (DC) input amplitude

We defined a simple code to input the current and time constant and visualize the plot .

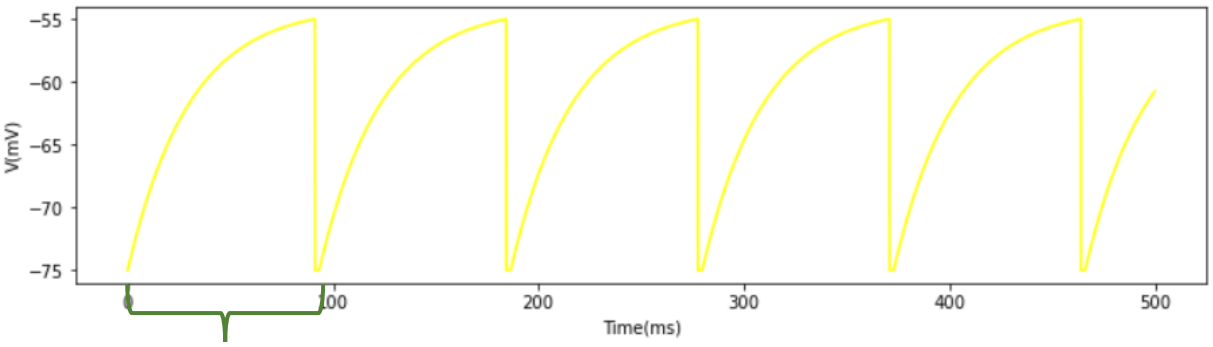
As we try different amounts of input currents with same time constant(tau=10),turns out around I=200pA is needed to reach the threshold & when DC current reaches almost 210pA we cross the threshold voltage.



- the membrane needs more time to reach the threshold after the reset ,so increasing the membrane time constant (slower membrane) cause the firing rate decrease.

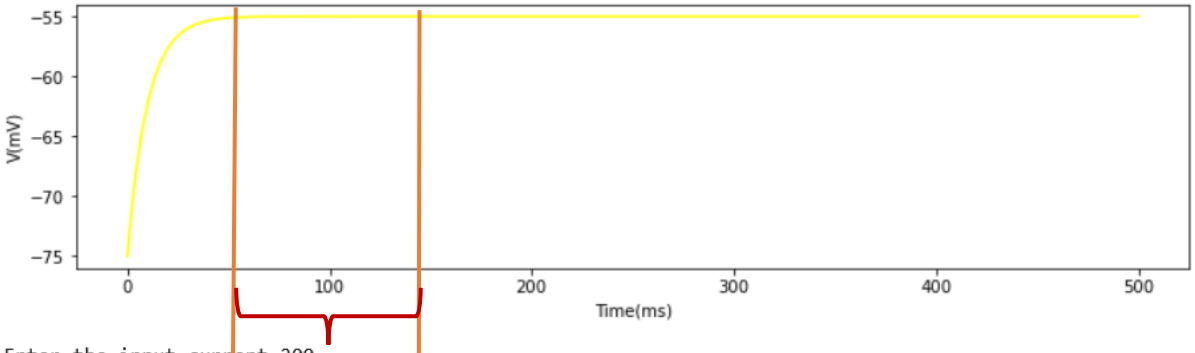


```
Enter the input current 210
Enter the time constant 30
```

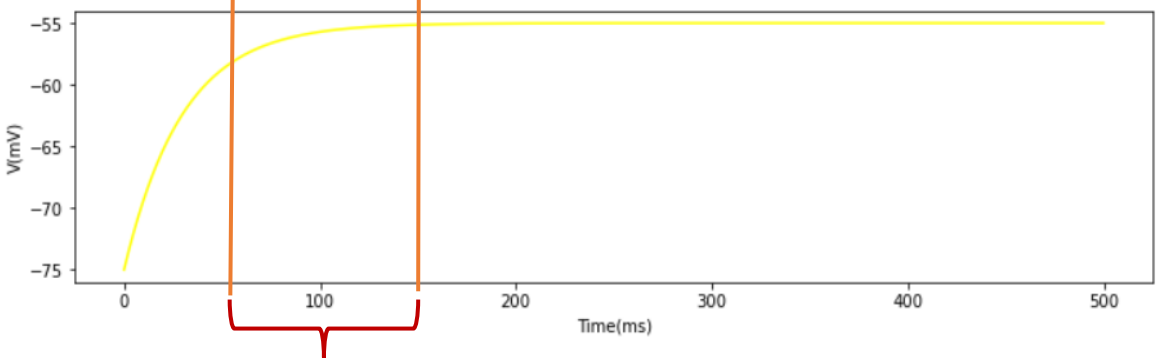


- in first picture (I=210pA, $\tau = 10$): 3spikes in 100ms
- in second picture (I=210pA, $\tau = 30$): 1spikes in 100ms

```
Enter the input current 200
Enter the time constant 10
```




```
Enter the input current 200
Enter the time constant 30
```



- third picture (I=200pA, $\tau = 10$): reach to threshold around 50ms
- forth picture (I=200pA, $\tau = 30$): reach to threshold around 150ms

Gaussian white noise (GWN) current

This code generates a time signal consisting of Gaussian white noise (GWN) with mean value "mu" and standard deviation "sig". The length of the signal is "Lt" time points, with the time increment between points specified as "dt".

The "np.random.seed()" function sets the seed for the random number generator used to generate the GWN. If a specific seed value "myseed" is given, it is used to set the seed. If "myseed" is not given, a random seed is chosen.

The line that generates the GWN is "(sig * np.random.randn(Lt) / np.sqrt(dt / 1000.)) + mu".

"np.random.randn(Lt)" generates an array of random numbers from a standard normal distribution with mean of 0 and standard deviation of 1.

This array is then multiplied by "sig", to obtain a GWN signal with the desired standard deviation.

The division by the square root of the time increment "dt" scaled by 1000 is to adjust the units of the noise to seconds.

Finally, the mean value "mu" is added to the signal.

Overall, this code generates a time signal of Gaussian white noise with specified mean and standard deviation

Analyzing GWN Effects on Spiking

1. Fluctuations can increase voltage and bring it closer to the threshold voltage ;so we need smaller input current because a part of path for spike to reach the threshold has already been covered!
2. The standard deviation represents of dispersal of irregularity of the spikes (or we can call it noise).By increasing sigma, the more range is expanded so we observe more irregularity of the spikes.

The interspike interval (ISI)

The interspike interval is the time between subsequent action potentials (also known as spikes) of a neuron, or a group average thereof. Action potentials are propagated along the axons of a neuron, to reach the nerve terminals, where they can trigger the release of chemical messengers to affect other neurons.

F-I Explorer for different sig_gwn

In the first part of code, we define sigma_affect function to obtain output rate spikes and then in the second part of the code, we show the data with plot.

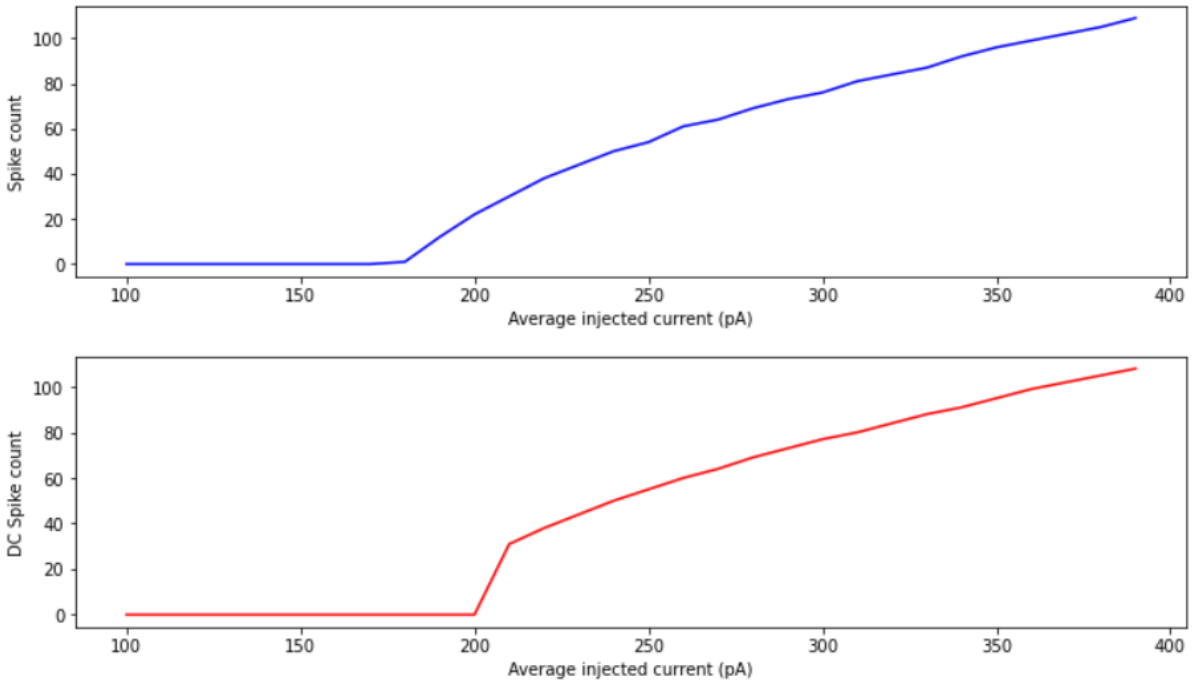
- ✓ In the function, first we set the default_pars to have constant variables .
- ✓ We use Ave_I to store a range of average input current that starts with 100pA and ends with 400pA(by step =10).
- ✓ We used np.zeros method to give SpikeCount & SpikeCountDC an array of zeros along Ave_I ‘s length.
- ✓ In the loop, variables are valued by using previous functions,in the end requested variables are returned.

```
def sigma_affect(sig_gwn):
  pars = default_pars(T=1000.)
  AveI = np.arange(100., 400., 10.)  #numpy.arange([start, ]stop, [step, ]dtype=None, *, like=None)
                                      #Return evenly spaced values within a given interval.
  SpikeCount = np.zeros(len(AveI))    #numpy.zeros(shape, dtype=float, order='C', *, like=None)
                                      #Return a new array of given shape and type, filled with zeros.
  SpikeCountDC = np.zeros(len(AveI))

  for i in range(len(AveI)):
      I_GWN = my_GWN(pars, mu=AveI[i], sig=sig_gwn, myseed=False)
      v, rec_spikes = run_LIF(pars, Iinj=I_GWN)
      SpikeCount[i] = len(rec_spikes)
      v_dc, rec_sp_dc = run_LIF(pars, Iinj=AveI[i])
      SpikeCountDC[i] = len(rec_sp_dc)
  return SpikeCount , SpikeCountDC , AveI
```

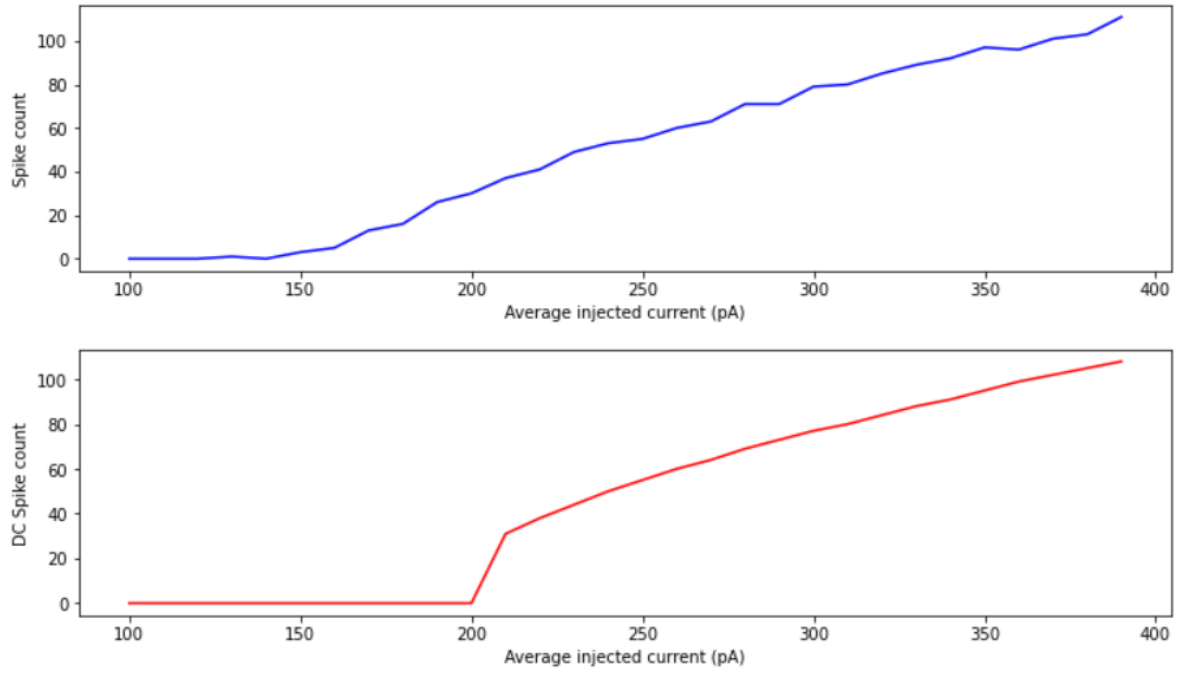
- ✓ In the second part of the code , we just get the sig_gwn from user and refer to the function and then we plot the F_I curve with obtained variables.

```
sig_gwn=int(input("Enter the input standard deviation"))
SpikeCount , SpikeCountDC , AveI= sigma_affect(sig_gwn)
#print(AveI)
plt.plot(AveI, SpikeCount, color='blue')
plt.ylabel('Spike count')
plt.xlabel('Average injected current (pA)')
plt.show()
plt.plot(AveI, SpikeCountDC, color='red')
plt.ylabel('DC Spike count')
plt.xlabel('Average injected current (pA)')
plt.show()
```

```
Enter the input standard deviation 1
```




```
Enter the input standard deviation 3
```



As we've guessed in the previous part, GWN can help reach to the threshold voltage with less input current; by increasing the standard deviation of GWN the F-I curve becomes more linear but with DC input, the curve doesn't change very much.

Compute CV_{ISI} values

We use spike_times as an argument, which is a vector (ndarray) containing the times at which the neuron fired; if at least two spikes happens ,we use np.diff method to Calculate the n-th discrete difference along the given axis and save the value as difference between spike's firing time; Else we put nothing(np.nan) in those variables.

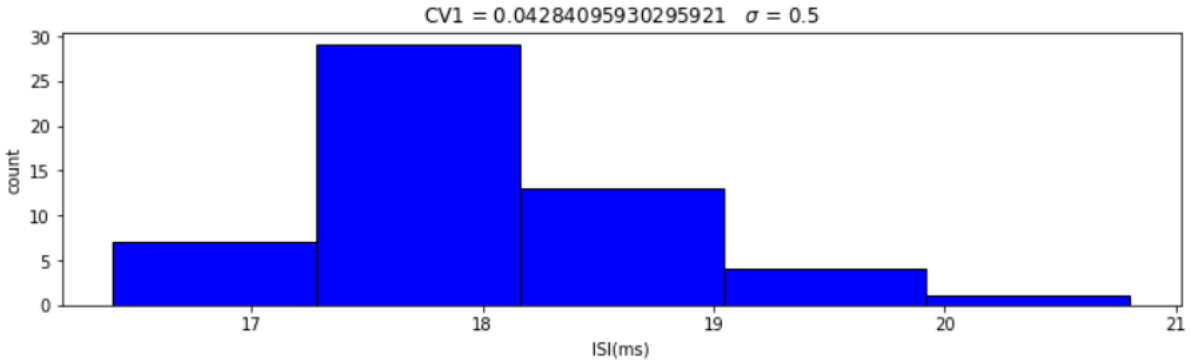
We can also calculate CV by the given formula:

$$\mathrm{CV_{ISI}} = \frac{std(\mathrm{ISI})}{mean(\mathrm{ISI})}$$

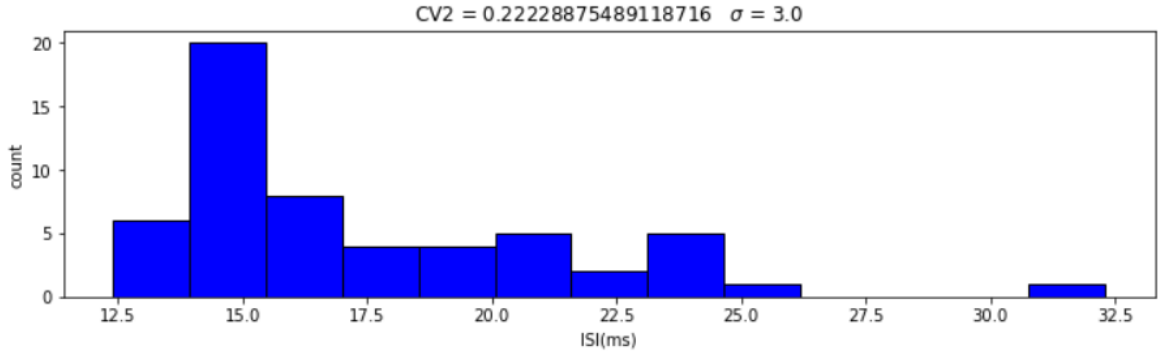
The rest of the code sets some parameters for a leaky integrate-and-fire (LIF) neuron model and runs the model twice, once with a low standard deviation for the input current (sig_gwn1 = 0.5) and once with a high standard deviation (sig_gwn2 = 3.0). The function my_GWN generates Gaussian white noise as the input current for the LIF model, and run_LIF executes the simulation. The isi_cv_LIF function is then called twice to compute the isi and cv for the two sets of spike times (sp1 and sp2). Finally, a histogram of isi1 is plotted with the title and axis labels that show the value of cv and the sigma.

mu_gwn = 250

```
[56]: <function matplotlib.pyplot.show(close=None, block=None)>
```




```
[30]: <function matplotlib.pyplot.show(close=None, block=None)>
```



Spike irregularity explorer for different sig_gwn

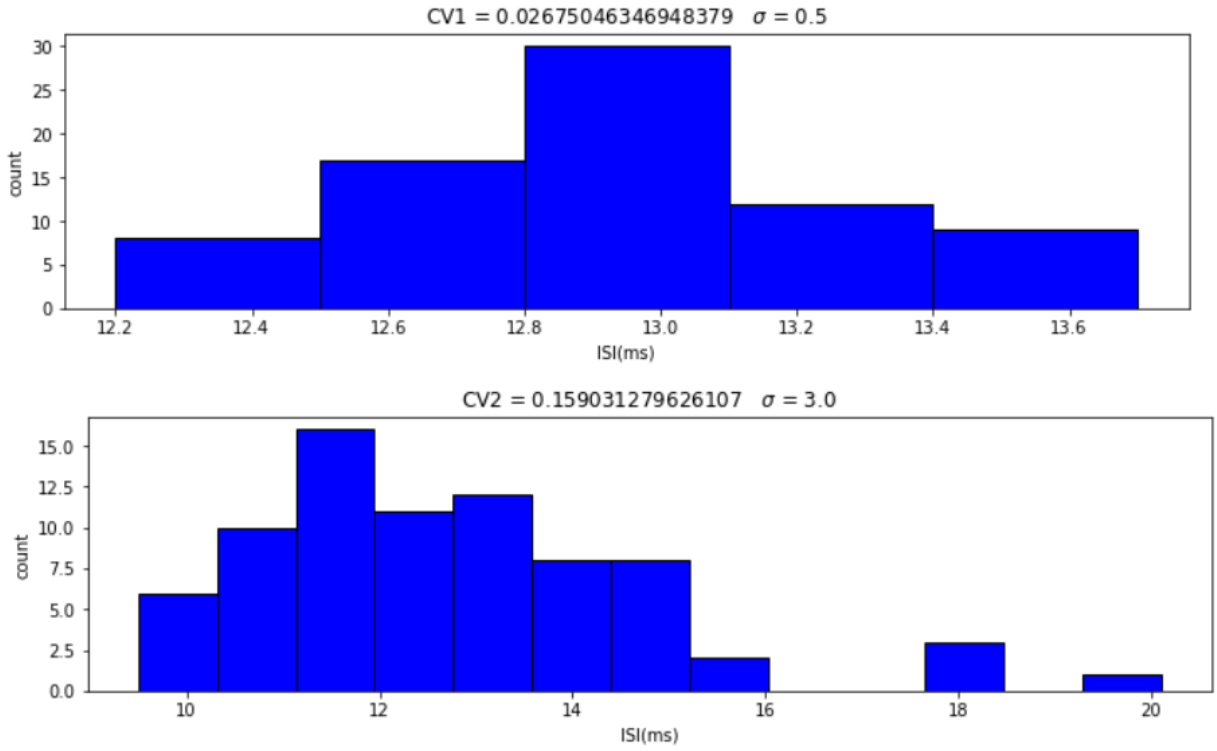
1)As we can see in "F-I Explorer for different sig_gwn" part, with DC current we have a strong non-linear F-I curve, but as we increaseσ it helps the curve becomes more linear.

2)We can change the mu in the last part and observe the affects;as we can see sometimes input mean voltage is higher than threshold voltage , so the only function a neuron can do is to reach the threshold then rest.
We can answer this question in different way; according to the below formula:

CV_{ISI}=std(ISI)/$mean$(ISI)

mean parameter has indirect relation with CV, in result as mean increase,less CV become.

Example :CV with mu_gwn = 300



3)CV is a constant parameter that just depends on two parameter in the previous part formula, and basically it depends directly on ISI which is the time between two spikes and it doesn't relate to any kind of physical quantity except time!

(found in research):In an LIF, high firing rates are achieved for high GWN mean. Higher the mean,higher the firing rate and lower the CV_ISI. So you will expect that as firing rate increases, spike irregularity decreases. This is because of the spike threshold.for a Poisson process there is no relationship between spike rate and spike irregularity.

Ornstein-Uhlenbeck Process

By using the Ornstein-Uhlenbeck process formula we can fill the blanket.

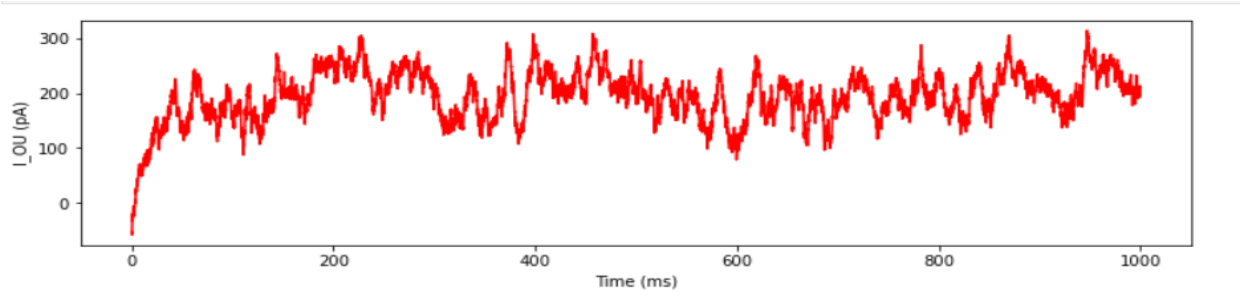
(noise[it+1] is the next random noise value) $\tau_\eta \frac{d}{dt}\eta(t) = \mu - \eta(t) + \sigma_\eta \sqrt{2\tau_\eta}\xi(t)$

We use run_LIF function to get v and then we draw the plot

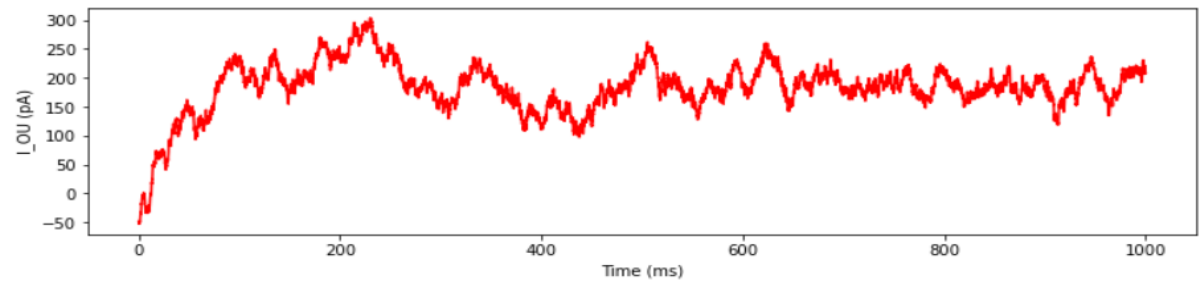
```
pars = default_pars(T=1000.)
pars['tau_ou'] = tau_ou=10  # [ms]
pars['mu_ou'] = mu_ou=200
pars['sig_ou'] = sig_ou=40

I_ou = my_OU(pars, mu_ou, sig_ou)

v, sp = run_LIF(pars, Iinj=I_ou)

plt.plot(pars['range_t'], I_ou, 'RED')
plt.xlabel('Time (ms)')
plt.ylabel('I_OU (pA)')
plt.show()
```

We can see as we increase time constant firing rate decrease as we expected, and spikes become more regular;so there's less range of color noise in the plot. It also makes CV constant smaller.

tau_ou=10



tau_ou=30



Extensions to Integrate-and-Fire models

Generalized Integrate-and-Fire models:

Generalized Leaky Integrate and Fire (GLIF) models are a simple point neuron models that have a striking ability to reproduce the spike times of real neurons. To explore how different phenomenological mechanisms contribute to spiking behavior, GLIF models of varying complexity are fit to electrophysiology data.

#The Hodgkin-Huxley model

We can use these formulas and information to fill the code:

- Conductances of the channels:

$$g_K(V) = \overline{g_K}\, n^4$$

$$g_{Na}(V) = \overline{g_{Na}}\, m^3 h$$

$\overline{g_K}$ and $\overline{g_{Na}}$ are the total (maximum) conductances of the channels

are hence ***voltage dependent***

$$-C_m \frac{dV}{dt} = g_L(V - E_L) + \overline{g_K}\, n^4(V - E_K) + \overline{g_{Na}}\, m^3 h(V - E_{Na}) - I_e$$

V = the voltage of neuron.

m = activation variable for Na-current.

h = inactivation variable for Na-current.

n = activation variable for K-current.

t = the time axis of the simulation

This code defines functions for the Hodgkin-Huxley model of neuron dynamics and then uses those functions to simulate the behavior of a neuron over time.

- ✓ The functions alphaM(), betaM(), alphaH(), betaH(), alphaN(), and betaN() calculate the rates of the opening and closing of different types of ion channels in the neuron, based on the neuron's membrane voltage V. These functions are based on empirical fits to experimental data.

✓ The function HH() simulates the behavior of the neuron by iterating over many small time steps dt. It initializes the simulation parameters, including the conductances gNa0, gK0, and gL0 of different ion channels in the neuron, the reversal potentials ENa and EK for sodium and potassium ions, respectively, and the resting membrane potential EL.

✓ The function then initializes arrays for the neuron's voltage V and the activation/inactivation parameters m, h, and n of the different ion channels. These arrays are then updated at each time step using the differential equations defined by the Hodgkin-Huxley model. The voltage V is updated based on the membrane currents generated by the different channels, and the activation/inactivation parameters are updated based on the voltage-dependent rates defined by the alphaM(), betaM(), alphaH(), betaH(), alphaN(), and betaN() functions.

✓ The function returns the arrays of the neuron's voltage V, and the activation/inactivation parameters m, h, and n, as well as the time-array t over which the simulation was performed. Overall, the HH() function calculates the evolution of the neuronal voltage and gating variables over time, based on the input current I0 and the duration of the simulation T0.

The last part is just for showing the diagrams for different variable:

```
def DrawPlot(t, variable, Name):
 plt.plot(t, variable, 'blue')
 plt.xlabel('Time (ms)')
 plt.ylabel(Name)
 plt.show()
```

```
I0 = 0
T0 = 100
pars = default_pars(T=1000)
#print(pars)
range_t = pars['range_t']
[V,m,h,n,t]=HH(I0,T0)
#print(V,m,h,n,t)
DrawPlot(range_t, V, 'V(v)')
DrawPlot(t, m, 'm')
DrawPlot(t, h, 'h')
DrawPlot(t, n, 'n')
```

Answering question with drawing the plots in the next page:

Final values(I0=0):

V = -65mv, m= -0.055, h=-0.59, n=-0.317

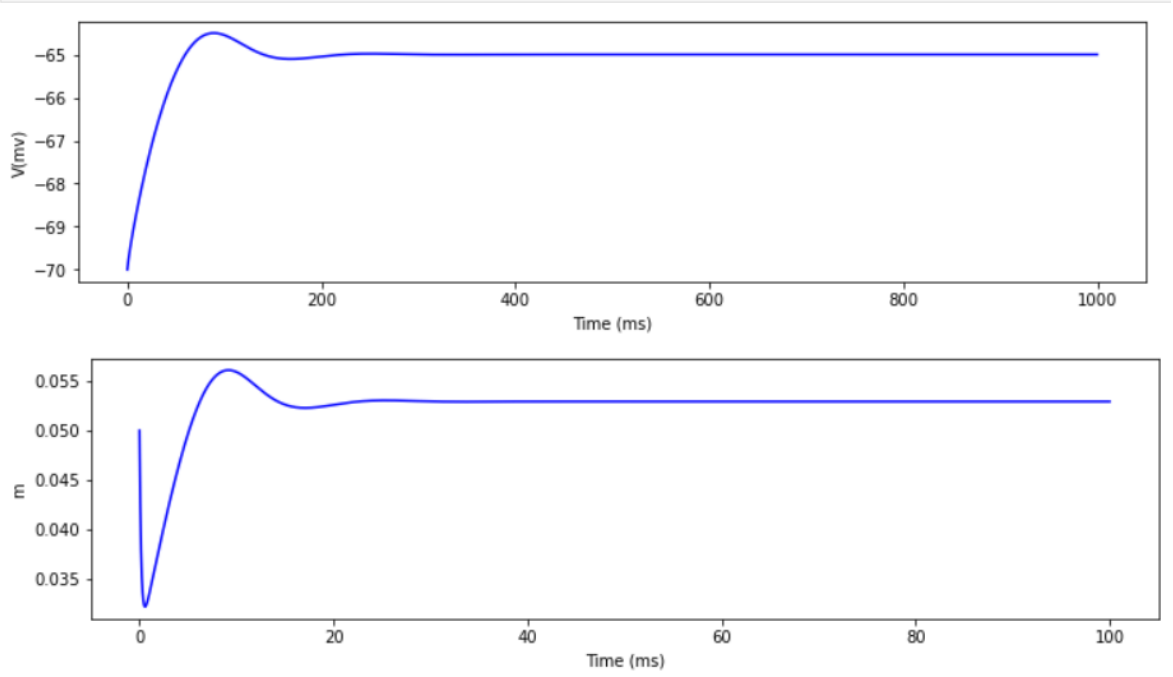
I0=0

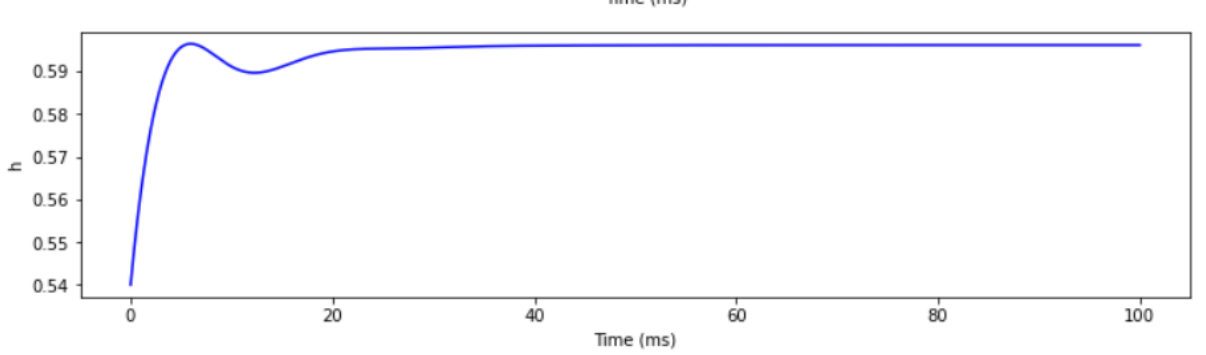
The resulting plot shows that the voltage initially stays at rest at -65 mV, but then exhibits small fluctuations around this value, which are caused by the random fluctuations in the ion channel gating variables. These fluctuations are also known as "spontaneous activity" and are a characteristic feature of neurons. However, without any external input (I0=0), the voltage does not reach the threshold required for generating an action potential.

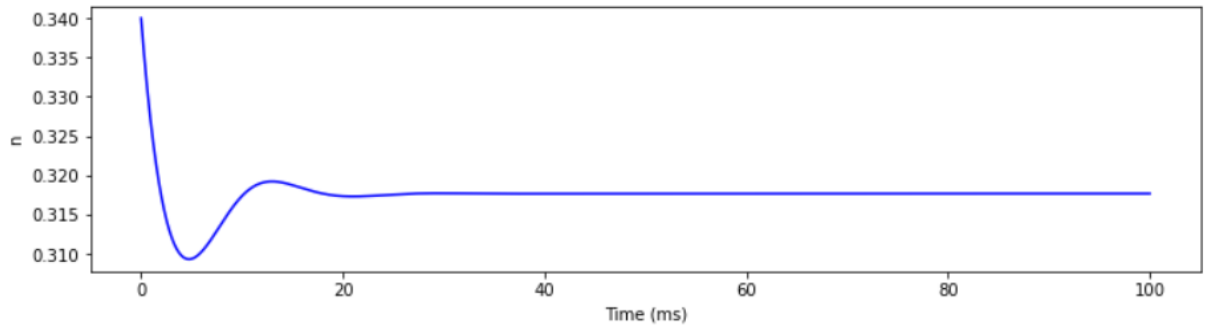
The variable 'm' is initially at 0.05 and increases rapidly to a peak value of around 0.06 within the first few milliseconds of the simulation. Then it starts to slowly decrease and reaches a steady-state value of around 0.055 after about 50 ms.This indicates that the sodium (Na+) channels are initially more open due to the depolarization of the membrane potential, but then they gradually become less active as the membrane potential increases and becomes more positive.

The variable 'h' is initially at 0.54 and increases rapidly to a steady-state value of around 0.6 within the first few milliseconds of the simulation. This indicates that the potassium (K+) channels are initially less active due to the depolarization of the membrane potential, but then they become more active as the membrane potential continues to increase and becomes more positive.

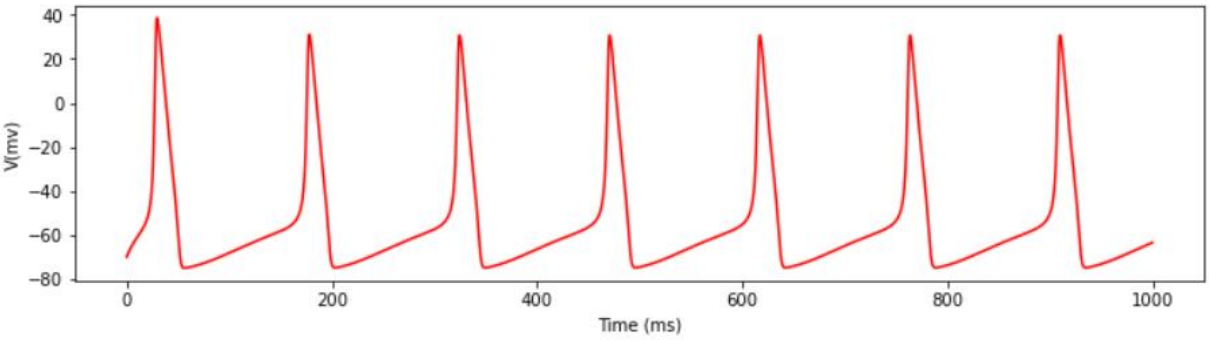
The variable 'n' is initially at 0.34 and slowly increases to a peak value of around 0.7 after about 10 ms. Then it starts to gradually decrease and reaches a steady-state value of around 0.317 after about 50 ms. This indicates that the potassium (K+) channels are initially less active due to the depolarization of the membrane potential, but then they become more active and help to repolarize the membrane potential.

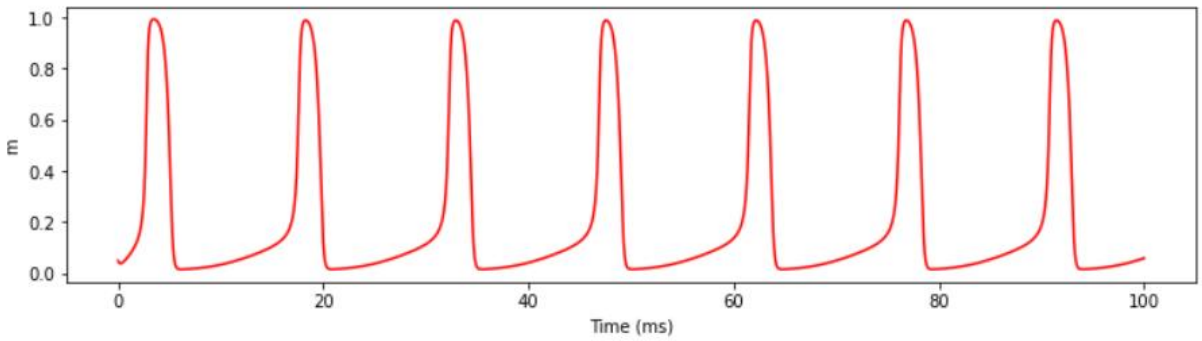



h
0.59
0.58
0.57
0.56
0.55
0.54
0
20
40
60
80
100
Time (ms)

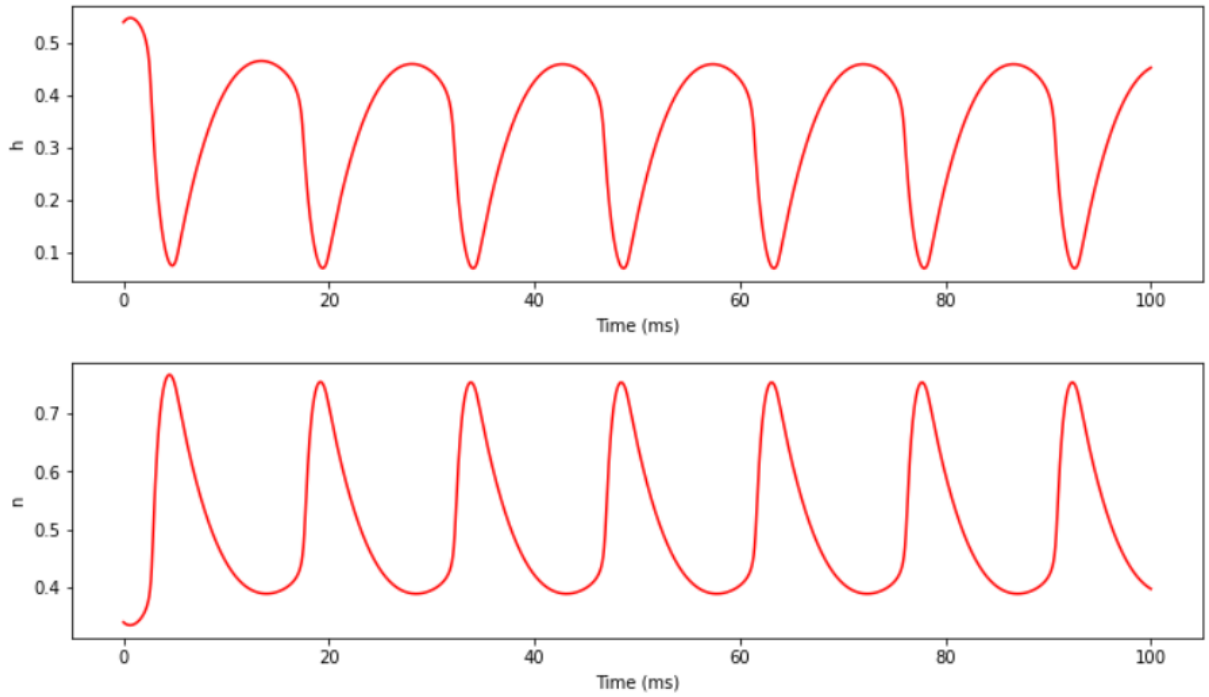

n
0.340
0.335
0.330
0.325
0.320
0.315
0.310
0
20
40
60
80
100
Time (ms)

I0=10


V(mv)
40
20
0
−20
−40
−60
−80
0
200
400
600
800
1000
Time (ms)


m
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Time (ms)

I0=10


h
0.5
0.4
0.3
0.2
0.1
0
20
40
60
80
100
Time (ms)

n
0.7
0.6
0.5
0.4
0
20
40
60
80
100
Time (ms)