```
#!/usr/bin/env python
# coding: utf-8

# <h1><center>Problem 2</center></h1>

# In[1]:


from IPython.display import HTML
HTML('''<script>
code_show=true;
function code_toggle() {
 if (code_show){
 $('div.input').hide();
 } else {
 $('div.input').show();
 }
 code_show = !code_show
}
$( document ).ready(code_toggle);
</script>
<a href="javascript:code_toggle()">
<button>Toggle Code</button></a>''')


# In[2]:


get_ipython().run_line_magic('matplotlib', 'inline')
# Import libraries
import numpy as np
import matplotlib.pyplot as plt


# #### Methods
#
# a.) We create an Integrate and Fire neuron using the equations and parameters defined in part a of the problem. The model was run with a step current injecion of 1 $nA$ for 50 $ms$ and its membrane potential, $V(t)$, plotted. Additionally, several sinusoidal inputs with frequencies 1 Hz, 2 Hz, 5 Hz, 10 Hz, 20 Hz, 50 Hz, and 100 Hz were tested on the model, while membrane potential was plotted. Spike count over each frequecies was also examined.
#
# b.) The same procedure was carried on the model described in part b. We also plot the response of threshold function, $U(t)$, alongside the membrance potential, $V(t)$.
#
# c.) A Two-Neuron oscillator was modelled using the parameters and equations given by part c. A constant asymmetric current input was given to the two neuron system, 1.1 $nA$ to neuron 1, and 0.9 $nA$ to neuron 2. Membrane potential, $V(t)$, was plotted for both neurons over 1500 $ms$.

# In[3]:


"""
    Define a current generator and Integrate-and-Fire class
"""

class CurrentGenerator:
    """
        A class defining a currrent generator
    """
    def __init__(self, I=1, It=50, It_start=10, freq=0, time_step=1):
        """
```

```
            Initialize current generator:
            (Current Amplitude)I = 1 nA
            (Time length of current) It = 50 ms
            (Time start of current) It_start = 10 ms
            (Frequency of current) freq = 0 (constant current)
            (Time Step of iteration) time_step = 1 ms
        """

        # Set parameters
        parameters = locals()
        for key in parameters:
            if not key == 'self':
                setattr(self, key, parameters[key])

        # create counter to determine current step
        self.counter = 0

        # Record current values
        self.I_vals = []

    def __call__(self):
        """
            Get next step of current generator
        """
        # set current injection start/duration
        if (self.counter > self.It_start and # Constant current
            self.counter <= self.It_start+self.It and
            self.freq == 0):
                I = self.I
        elif (self.counter > self.It_start and # Sinusoidal current
            self.counter <= self.It_start+self.It and
            self.freq != 0):
                I = self.I*np.sin(2*np.pi*self.freq*(self.counter*self.time_step/1000))
        else: # Zero Current
            I = 0

        # increment counter
        self.counter += 1

        # add current value to list
        self.I_vals.append(I)

        # return the current value
        return I

class IntegrateAndFire:
    """
        A class representing the integrate and fire neuron model
    """
    def __init__(self, cg=CurrentGenerator(), R=10, C=1, Vthr=5, Vspk=70):
        """
            Defined Constants/Initial values:
            R = 10 MOhm
            C = 1 nF
            Vthr = 5 mV
            Vspk = 70 mV
        """

        # Set parameters
        parameters = locals()
        for key in parameters:
            if not key == 'self':
                setattr(self, key, parameters[key])
```

```python
        # Set initial voltage
        self.V = 0

        # Create dict to store plot data
        self.plot_store = {}
        self.plot_store['V'] = []

        # set reset flag so we can reset voltage
        self.reset = False

        # keep track of spike counts
        self.spike_count = 0

    def new_voltage(self, V, I):
        """
            Calculates new voltage using Euler's method
        """
        return self.V + (I - (V/self.R))/self.C

    def set_voltage(self,V):
        """
            Sets appropriate voltage compared to threshold
        """
        # check if voltage less than threshold
        if V < self.Vthr:
            return V
        else: # if over threshold, set voltage to Vspk, next V will be 0
            self.reset = True
            self.spike_count += 1 # add to spike count
            return self.Vspk

    def step(self):
        """
            Runs one step of the model
        """
        # get the current from the generator
        I = self.cg()

        # reset voltage if flag set
        if self.reset:
            self.V = 0

            # "reset" reset flag
            self.reset = False
        else:
            # calculate and set new voltage
            self.V = self.set_voltage(self.new_voltage(self.V, I))

        # append voltage to list
        self.plot_store['V'].append(self.V)


# <h4><center>(i) Integrate and Fire</center></h4>

# In[4]:


"""
    Run model
"""
def run_model(model, title, iterations=100, show_input=False, create_legend=False):
    # run model for iterations
```

```
    for _ in range(iterations):
        model.step()

    # set time
    try:
        time_step = model.cg.time_step
    except AttributeError: # Just set to 1 ms, if the cg attribute doesn't exist...
        time_step = 1
    time = [i*time_step for i in range(iterations)]

    # plot stuff
    plt.figure(figsize=(16,8))
    if show_input:
        # Setup axes
        ax1 = plt.axes()
        ax1.set_xlabel('Time (ms)')
        ax1.set_ylabel('Voltage (mV)')
        ax2 = ax1.twinx()
        ax2.set_ylabel('Current (nA)', color='m')
        ax2.tick_params('y', colors='m')

        # Loop through each key and plot
        for key in model.plot_store:
            ax1.plot(time, model.plot_store[key], label=key)

        # Plot current
        ax2.plot(time, model.cg.I_vals, 'm:', label='Input')

        # create legend if set
        if create_legend:
            ax1.legend()
    else:
        # Loop through each key and plot
        for key in model.plot_store:
            plt.plot(time, model.plot_store[key], label=key)
        # Label Axes
        plt.xlabel('Time (ms)')
        plt.ylabel('Voltage (mV)')

        # create legend if set
        if create_legend:
            plt.legend()

    # set title and show plots
    plt.title(title)
    plt.show()

# create integrate and fire model with defaults and run model
run_model(IntegrateAndFire(), 'Integrate and Fire Neuron', show_input=True)


# <h4><center>(ii) Integrate and Fire at Various Sinusoidal Frequencies</center></h4>

# In[5]:


"""
    Run various sinusoidal currents on integrate and fire model
"""
def run_sinusoidal_models(model, title, I=1, create_legend=False):
    # set frequncies to run
    frequency = [1, 2, 5, 10, 20, 50, 100]
```

```python
    # create model with sinusoidal currents for each frequency and run
    sinusoid_model = []
    for f in frequency:
        sinusoid_model.append(model(cg=CurrentGenerator(I=I, It_start=0, It=1000, freq=f)))
        run_model(
            sinusoid_model[-1],'{} {}Hz'.format(title,f),
            iterations=1000,
            show_input=True,
            create_legend=create_legend
        )

    # plot spike count vs. frequency
    spike_counts = [m.spike_count for m in sinusoid_model]
    plt.figure(figsize=(16,8))
    plt.plot(frequency, spike_counts)
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Spike Count')
    plt.title('Frequency vs. Spike Count')
    plt.show()

# run integrate and fire model
run_sinusoidal_models(IntegrateAndFire, 'Integrate and Fire')


# <h4><center>(iii) Part b Neuron</center></h4>

# In[6]:


"""
    Model part b
"""

class ModelB:
    """
        A class representing the model described in part b
    """
    def __init__(self, cg=CurrentGenerator(), a=0.02, b=0.2, c=-65, d=8, v =-65):
        """
            Defined Constants/Initial values:
            a = 0.02
            b = 0.2
            c = -65
            d = 8
            v = -65
            u = b*v
        """

        # Set parameters
        parameters = locals()
        for key in parameters:
            if not key == 'self':
                setattr(self, key, parameters[key])

        # Set initial u
        self.u = b*v

        # Create dict to store plot data
        self.plot_store = {}
        self.plot_store['u'] = []
        self.plot_store['v'] = []

        # set reset flag so we can reset voltage
```

```
        self.reset = False

        # keep track of spike counts
        self.spike_count = 0

    def new_params(self, u, v, I):
        """
            Calculates new parameters using Euler's method

        """
        # Calculate derivatives
        v_prime = (0.04*v**2 + 5*v + 140 - u + I)
        u_prime = (self.a*(self.b*v - u))

        # Calculate new v and u
        v_new = v + v_prime
        u_new = u + u_prime

        return u_new, v_new

    def set_params(self, u, v):
        """
            Sets appropriate voltage compared to threshold
        """
        # check if voltage is >equal to 30
        if v >= 30:
            self.reset = True # clamp to set values on next iteration
            self.spike_count += 1 # add to spike count
            return self.u, 30
        else:
            return u, v

    def step(self):
        """
            Runs one step of the model
        """
        # get the current from the generator
        I = self.cg()

        # set params to set values if flag set
        if self.reset:
            self.u = self.u + self.d
            self.v = self.c

            # "reset" reset flag
            self.reset = False
        else:
            # calculate and set new voltage
            self.u, self.v = self.set_params(*self.new_params(self.u, self.v, I))

        # append voltage to list
        self.plot_store['u'].append(self.u)
        self.plot_store['v'].append(self.v)

# create model b with I = 10 nA for 50 ms and run model
run_model(
    ModelB(cg=CurrentGenerator(I=10, It=50, It_start=10)),
    'Model b Neuron', iterations=100, show_input=True, create_legend=True)


# <h4><center>(iv) Part b Neuron at Various Sinusoidal Frequencies</center></h4>

# In[7]:
```

```python
"""
    Run different frequencies
"""
# Run for different sinusoidal inputs
run_sinusoidal_models(ModelB, 'Model b', I=10, create_legend=True)


# <h4><center>(v) Two Neuron Oscillator</center></h4>

# In[8]:


"""
    Model Two Neuron Oscillator
"""
class TwoNeuronOscillator:
    """
        A class representing a Two Neuron Oscillator
    """
    def __init__(self, cg1=CurrentGenerator(I=1.1, It_start=0, It=1500),
                 cg2=CurrentGenerator(I=0.9, It_start=0, It=1500),
                 C=1, R=10, Vrest=0, Vspk=70, tauthresh=50, Einh=-15,
                 tausyn=15, gpeak=0.1):
        """
            Defined Constants/Initial values:
            (membrane capacitance) C = 1 nF
            (membrane resistance) R = 10 MOhms
            (resting membrane potential) Vrest = 0 mV
            (action potential amplitude) Vspk = 70 mV
            (threshold time constant) tauthresh = 50 ms
            (synaptic reversal potential) Einh = -15 mV
            (synaptic time constant) tausyn = 15 ms
            (peak synaptic conductance) gpeak = 0.1 uS
        """
        # Set parameters
        parameters = locals()
        for key in parameters:
            if not key == 'self':
                setattr(self, key, parameters[key])

        # Create parameters for each neuron
        self.neuron = []
        for n in range(2):
            self.neuron.append({})
            self.neuron[n]['v'] = Einh
            self.neuron[n]['theta'] = Vrest
            self.neuron[n]['z'] = 0.1
            self.neuron[n]['g'] = 0.1

        # Create dict to store plot data
        self.plot_store = {}
        self.plot_store['Neuron 1'] = []
        self.plot_store['Neuron 2'] = []

        # set reset flags so we can reset voltage
        self.reset1 = False
        self.reset2 = False

    def new_params(self, n, I):
        """
            Calculates new parameters using Euler's method
```

```
        """
        # Get other neuron index
        on = {1: 0, 0: 1}[n]

        # Calculate derivatives
        dvdt = (1/self.C)*((-self.neuron[n]['v']/self.R) - self.neuron[n]['g']*(self.neuron[n]
['v'] - self.Einh) + I)
        dthetadt = (-self.neuron[n]['theta'] + self.neuron[n]['v'])/self.tauthresh
        dzdt = (-self.neuron[n]['z']/self.tausyn) + (self.gpeak/(self.tausyn/
np.exp(1)))*(self.neuron[on]['v']==self.Vspk)
        dgdt = (-self.neuron[n]['g']/self.tausyn) + self.neuron[n]['z']

        # return derivatives
        return (dvdt, dthetadt, dzdt, dgdt)

    def apply_params(self, n, dvdt, dthetadt, dzdt, dgdt):
        """
            Apply derivatives to current values
        """
        # Calculate new values
        self.neuron[n]['v'] = self.neuron[n]['v'] + dvdt
        self.neuron[n]['theta'] = self.neuron[n]['theta'] + dthetadt
        self.neuron[n]['z'] = self.neuron[n]['z'] + dzdt
        self.neuron[n]['g'] = self.neuron[n]['g'] + dgdt

    def set_params(self):
        """
            Sets appropriate voltage compared to threshold
        """
        # For each neuron, check voltage relative to threshold
        if self.neuron[0]['v'] >= self.neuron[0]['theta']:
            self.neuron[0]['v'] = self.Vspk
            self.reset1 = True
        if self.neuron[1]['v'] >= self.neuron[1]['theta']:
            self.neuron[1]['v'] = self.Vspk
            self.reset2 = True

    def step(self):
        """
            Runs one step of the model
        """
        # get the current from the generator
        I1 = self.cg1()
        I2 = self.cg2()

        # Calculate derivatives then update values
        derivative1 = self.new_params(0, I1)
        derivative2 = self.new_params(1, I2)
        self.apply_params(0, *derivative1)
        self.apply_params(1, *derivative2)

        # set params to reset values if flag set
        # For Neuron 1
        if self.reset1:
            # "reset" reset flag
            self.reset1 = False
            # set to baseline voltage
            self.neuron[0]['v'] = self.Einh

        # For Neuron 2
        if self.reset2:
            # "reset" reset flag
```

```
            self.reset2 = False
            # set to baseline
voltage
            self.neuron[1]['v'] = self.Einh

        # Check against thresholds
        self.set_params()

        # append parameters to list
        self.plot_store['Neuron 1'].append(self.neuron[0]['v'])
        self.plot_store['Neuron 2'].append(self.neuron[1]['v'])

# run Two neuron model
run_model(TwoNeuronOscillator(), 'Two Neuron Oscillator', iterations=1500, create_legend=True)


# #### Discussion
#
# a.) The Integrate and Fire neuron is a simpler model compared to the Hodgkin-Huxley model.
It provides an approximate model of the spiking behavior of neurons, and can be useful when
one wants to model the number of spikes a neuron responds to given a current injection input.
As can be seen by the various sinusoidal current inputs. The neuron model only fires when both
current intensity and it's duration are sufficient. This can easily be seen at the higher
frequecy current inputs, where the current amplitude is kept the same, but the neuron still
does not fire since the duration of the stimulus current is so short. Modeling spike count vs.
current input, we easily see that the Integrate and Fire neuron is a low-pass filter.
#
# b.) The part b neuron has similar behavior to the Integrate and Fire neuron. However, the
introduction of the threshold function, $U(t)$, seems to slow the firing rate of the neuron,
as it requires time for the threshold function to decrease before firing again. This attempts
to mimic the repolarization phase of the neuron. The spike count vs. current input plot shows
that this neuron model seems to be similar to a band-pass filter.
#
# c.) The Two-neuron oscillator model shows two neurons connected in mutual inhibitory
relationship. This means that the firing of one neuron inhibits the firing of the other neuron
and vice-versa. This inhibitory relationship is counteracted by the threshold function, $
\theta(t)$, which causes an inhibitory effect on the firing of the neuron the more frequently
it fires. Thus, the oscillatory behavior of the two-neuron model can be seen as a balancing of
these two forces. Since, neuron 1 is stimulated with a higher initial current, it takes the
opportunity to fire first, inhibiting neuron 2. This firing ceases after approximately 200 ms,
at which it's threshold function prevents it from firing any further. This decreases the
inhibitory effect on neuron 2 and allows it to fire which inhibits neuron 1. Neuron 2
continues to fire until it's threshold function prevents it from firing any further, and the
cycle from neuron 1 repeats yet again.
```