SYDE 556-750 Assignment 2

Peter Duggins, Student ID 20610432

February 22, 2016

Assignment 2 Details

- Due Date: Feb 22th: Assignment #2 (due at midnight)
- Total marks: 20 (20% of final grade)
- Late penalty: 1 mark per day
- It is recommended that you use a language with a matrix library and graphing capabilities. Two main suggestions are Python and MATLAB.
- Do not use any code from Nengo

1 Generating a random input signal

1.1 Gaussian white noise

Create a function called that generates a randomly varying $x(t)$ signal chosen from a white noise distribution. Call it 'generate_signal' and ensure that it returns $x(t)$ and $X(\omega)$. The inputs to the function are:

- `T`: the length of the signal in seconds
- `dt`: the time step in seconds
- `rms`: the root mean square power level of the signal. That is, the resulting signal should have $\sqrt{\frac{1}{T}\int x(t)^2 dt} = rms$
- `limit`: the maximum frequency for the signal (in Hz)
- `seed`: the random number seed to use (so we can regenerate the same signal again)
- $\Delta\omega$ will be $2\pi/T$
- To keep the signal real, $X(\omega) = X(-\omega)^*$ (the complex conjugate: the real parts are equal, and the imaginary parts switch sign)
- When randomly generating $X(\omega)$ values, sample them from a Normal distribution $N(\mu = 0, \sigma = 1)$. Remember that these are complex numbers, so sample twice from the distribution; once for the real component and once for the imaginary.
- To implement the `limit`, set all $X(\omega)$ components with frequencies above the limit to 0
- To implement the `rms`, generate the signal, compute its RMS power ($\sqrt{\frac{1}{T}\int x(t)^2 dt} = rms$) and rescale so it has the desired power.

These are the helper functions needed I wrote for the assignment.

To find α and J^{bias}: In the general case, α and J^{bias} can be calculated from two known points, $(x_1, a_1), (x_2, a_2)$ and the standard LIF approximation $a(J) = \frac{1}{\tau_{ref} - \tau_{RC} ln(1-\frac{1}{J})}$:

- $J^{bias} = (\frac{1}{1-\frac{x_2 \cdot e}{x_1 \cdot e}}) * (\frac{1}{1-e^{\frac{\tau_{ref}-a_2^{-1}}{\tau_{rc}}}} - \frac{x_2 \cdot e}{x_1 \cdot e} * \frac{1}{1-e^{\frac{\tau_{ref}-a_1^{-1}}{\tau_{rc}}}})$
- $\alpha = (\frac{1}{x_2 \cdot e}) * (\frac{1}{1-e^{\frac{\tau_{ref}-a_2^{-1}}{\tau_{rc}}}} - J^{bias})$

These equations simplify in the case of $x_1 \cdot e = 0$ to

- $J^{bias} = \frac{1}{1-e^{\frac{\tau_{ref}-a_1^{-1}}{\tau_{rc}}}}$
- $\alpha = (\frac{1}{x_2 \cdot e}) * (\frac{1}{1-e^{\frac{\tau_{ref}-a_2^{-1}}{\tau_{rc}}}} - J^{bias})$

```
In [119]: %pylab inline
          import numpy as np
          import matplotlib.pyplot as plt
          plt.rcParams['lines.linewidth'] = 4
          plt.rcParams['font.size'] = 24

          class spikingLIFneuron():

              def __init__(self,x1_dot_e,x2_dot_e,a1,a2,encoder,tau_ref,tau_rc):
                  self.x1_dot_e=float(x1_dot_e)
                  self.x2_dot_e=float(x2_dot_e)
                  self.a1=float(a1)
                  self.a2=float(a2)
                  self.e=encoder
                  self.tau_ref=tau_ref
                  self.tau_rc=tau_rc
                  self.Jbias=0.0
                  self.alpha=0.0
                  self.V=0.0
                  self.dVdt=0.0
                  self.stimulus=[]
                  self.spikes=[]
                  self.Vhistory=[]

                  #alpha and Jbias calculation
                  if self.x1_dot_e==0:
                      self.Jbias=1/(1-np.exp((self.tau_ref - 1/self.a1)/self.tau_rc))
                      self.alpha=(1/self.x2_dot_e)*\
                              (1/(1-np.exp((self.tau_ref - 1/self.a2)/self.tau_rc)) - self.Jbias)
                  elif self.x2_dot_e==0:
                      self.Jbias=1/(1-np.exp((self.tau_ref - 1/self.a2)/self.tau_rc))
                      self.alpha=(1/self.x1_dot_e)*\
                          (1/(1-np.exp((self.tau_ref - 1/self.a1)/self.tau_rc)) - self.Jbias)
                  #general case
                  else:
                      self.Jbias=(1/(1-self.x2_dot_e/self.x1_dot_e))-\
                        1/(1-np.exp((self.tau_ref - 1/self.a2)/self.tau_rc))-\
                        (self.x2_dot_e/self.x1_dot_e)*\
                        1/(1-np.exp((self.tau_ref - 1/self.a1)/self.tau_rc))
```

```
            self.alpha=(1/self.x2_dot_e)*\
              (1/(1-np.exp((self.tau_ref - 1/self.a2)/self.tau_rc)) - self.Jbias)

    def set_spikes(self,stimulus,T,dt):
        self.stimulus=stimulus #an array
        self.spikes=[]
        self.Vhistory=[]
        ref_window=int(self.tau_ref/dt) #number of timesteps in refractory period

        for t in range(len(stimulus)):
            self.J=self.alpha*np.dot(self.stimulus[t],self.e) + self.Jbias
            self.dVdt=((1/self.tau_rc)*(self.J-self.V))
            #check if there have been spikes in the last tau_rc seconds
            for h in range(ref_window):
                if len(self.spikes) >= ref_window and self.spikes[-(h+1)] == 1:
                    self.dVdt=0     #if so, voltage isn't allowed to change
            #Euler's Method Approximation V(t+1) = V(t) + dt *dV(t)/dt
            self.V=self.V+dt*self.dVdt
            if self.V >= 1.0:
                self.spikes.append(1)   #a spike
                self.V=0.0    #reset
            else:
                self.spikes.append(0)   #not a spike
                if self.V < 0.0: self.V=0.0
            self.Vhistory.append(self.V)

    def get_spikes(self):
        return self.spikes

def get_decoders_smoothed(spikes,h,x):

    S=len(x)
    #have to truncate from 'full' here, mode='same' messes up last half of list
    A_T=np.array([np.convolve(s,h,mode='full')[:len(spikes[0])] for s in spikes])
    A=np.matrix(A_T).T
    x=np.matrix(x).T
    upsilon=A_T*x/S
    gamma=A_T*A/S
    d=np.linalg.pinv(gamma)*upsilon
    return d

def get_estimate_smoothed(spikes,h,d):

    #have to truncate from 'full' here, mode='same' messes up last half of list
    xhat=np.sum([d[i]*np.convolve(spikes[i],h,mode='full')\
             [:len(spikes[i])] for i in range(len(d))],axis=0)
    xhat=xhat.T
    return xhat

def generate_signal(T,dt,rms,limit,seed,distribution='uniform'):

    #first generate X(w), with the specified constraints, then use an inverse fft to get x_t
    rng=np.random.RandomState(seed=seed)
    t=np.arange(int(T/dt))*dt
```

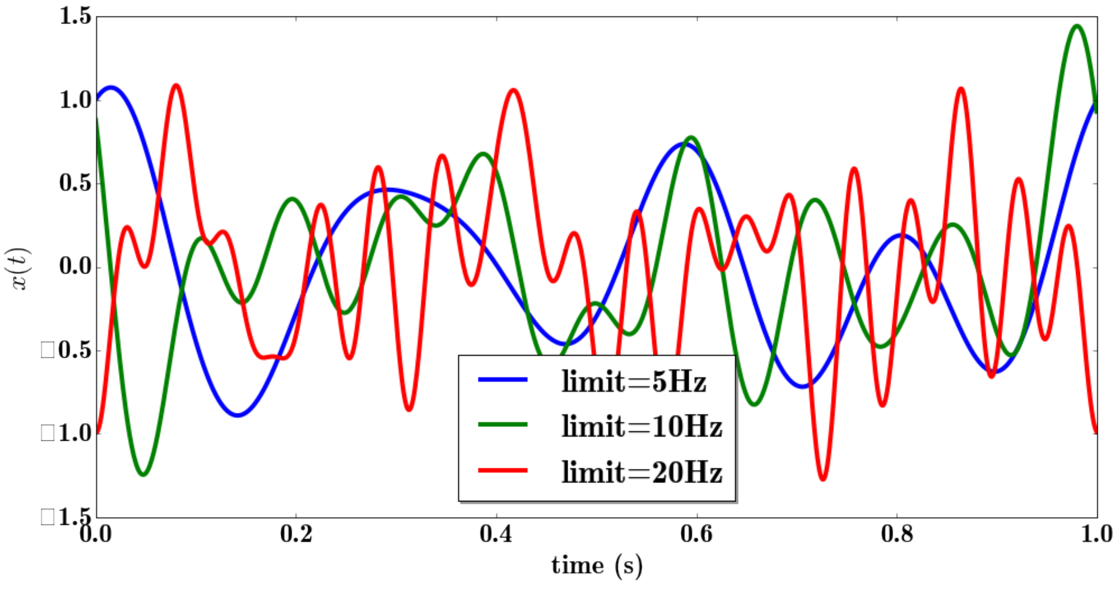
```
            delta_w = 2*np.pi/T #omega stepsize
            w_vals = np.arange(-len(t)/2,0,delta_w) #make half of X(w), those with negative freq
            w_limit=2*np.pi*limit #frequency in radians
            bandwidth=2*np.pi*limit #bandwidth in radians
            x_w_half1=[]
            x_w_half2=[]

            for i in range(len(w_vals)):    #loop over frequency values
                if distribution=='uniform':
                    #if |w| is within the specified limit, generate a coefficient
                    if abs(w_vals[i]) < w_limit:
                        x_w_i_real = rng.normal(loc=0,scale=1)  #mean=0, sigma=1
                        x_w_i_im = rng.normal(loc=0,scale=1)
                        x_w_half1.append(x_w_i_real + 1j*x_w_i_im)
                        #make the 2nd half of X(w) with complex conjugates
                        x_w_half2.append(x_w_i_real - 1j*x_w_i_im)

                elif distribution=='gaussian':
                    #draw sigma from a gaussian distribution
                    sigma=np.exp(-np.square(w_vals[i])/(2*np.square(bandwidth)))
                    if sigma > np.finfo(float).eps: #distinguishable from zero
                        x_w_i_real = rng.normal(loc=0,scale=sigma)
                        x_w_i_im = rng.normal(loc=0,scale=sigma)
                        x_w_half1.append(x_w_i_real + 1j*x_w_i_im)
                        x_w_half2.append(x_w_i_real - 1j*x_w_i_im)

            #zero pad the positive and negative amplitude lists, so each is len(samples/2)
            x_w_pos=np.hstack((x_w_half2[::-1],np.zeros(len(t)/2-len(x_w_half2))))
            x_w_neg=np.hstack((np.zeros(len(t)/2-len(x_w_half1)),x_w_half1))
            #assemble the symmetric X(w) according to numpy.fft documentation
            #amplitudes corresponding to [w_0, w_pos increasing, w_neg increasing]
            x_w=np.hstack(([0+0j],x_w_pos,x_w_neg))
            x_t=np.fft.ifft(x_w)
            #normalize time and frequency signals using RMS
            true_rms=np.sqrt(dt/T*np.sum(np.square(x_t)))
            x_t = x_t*rms/true_rms
            x_w = x_w*rms/true_rms

            #return real part of signal to avoid warning, but I promise they are less than e-15
            return x_t.real, x_w
```

```
Populating the interactive namespace from numpy and matplotlib
```

1.1.1 Plot $x(t)$ for three randomly generated signals with `limit` at 5, 10, and 20Hz. For each of these, `T`=1,`dt`=0.001, and `rms`=0.5.

```
In [120]: def one_pt_one_a():

            T=1
            dt=0.001
            rms=0.5
            limit=10
            seed=1
            t=np.arange(int(T/dt)+1)*dt
```

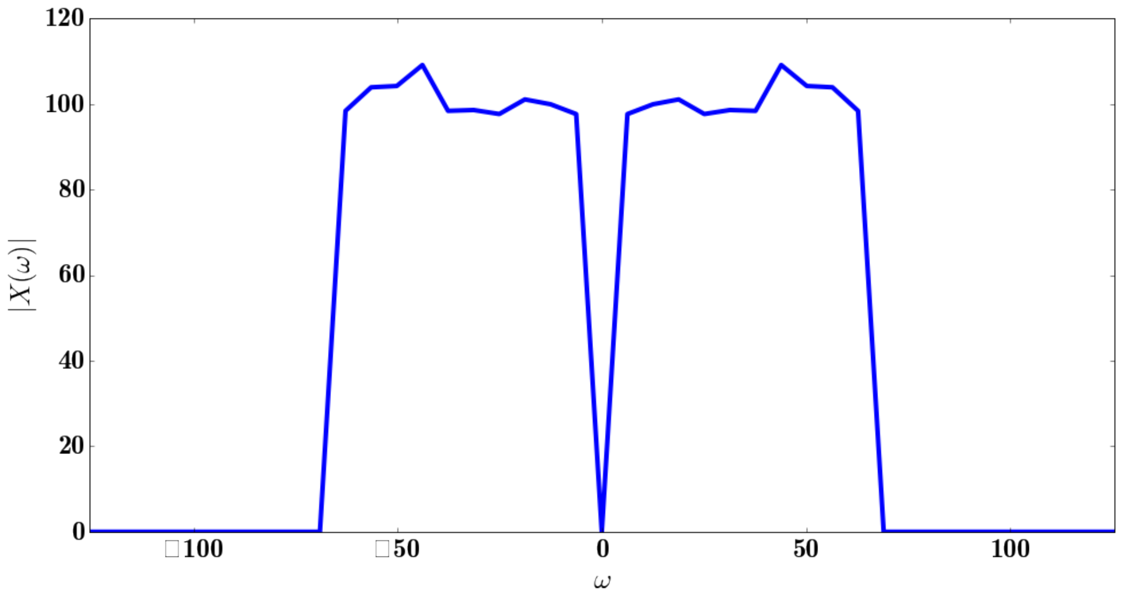
```
        limits=[5,10,20]
        x_t_list=[]
        for i in range(len(limits)):
            seed=i
            limit=limits[i]
            x_ti, x_wi = generate_signal(T,dt,rms,limit,seed,'uniform')
            x_t_list.append(x_ti)

        fig=plt.figure(figsize=(16,8))
        ax=fig.add_subplot(111)
        for i in range(len(limits)):
            ax.plot(t,x_t_list[i],label='limit=%sHz' %(limits[i]))
        ax.set_xlabel('time (s)')
        ax.set_ylabel('$x(t)$')
        legend=ax.legend(loc='best',shadow=True)
        plt.show()

    one_pt_one_a()
```



1.1.2 Plot the average $|X(\omega)|$ (the norm of the Fourier coefficients) over 100 signals generated with `T`=1, `dt`=0.001, `rms`=0.5, and `limit`=10 (each of these 100 signals should have a different `seed`). The plot should have the x-axis labeled (ω in radians) and the average $|X|$ value for that ω on the y-axis.

```
In [121]: def one_pt_one_b():

              T=1
              dt=0.001
              rms=0.5
              limit=10
              avgs=100
```

```
        x_w_list=[]
        for i in range(avgs):
            seed=i
            x_ti, x_wi = generate_signal(T,dt,rms,limit,seed,'uniform')
            x_w_list.append(np.abs(x_wi))
        x_w_avg=np.average(x_w_list,axis=0)
        w_vals=np.fft.fftfreq(len(x_w_avg))*2*np.pi/dt
        w_limit=2*np.pi*limit

        fig=plt.figure(figsize=(16,8))
        ax=fig.add_subplot(111)
        ax.plot(np.sort(w_vals),np.fft.fftshift(x_w_avg))
        ax.set_xlabel('$\omega$')
        ax.set_ylabel('$|X(\omega)|$')
        ax.set_xlim(-w_limit*2, w_limit*2)
        plt.show()

    one_pt_one_b()
```

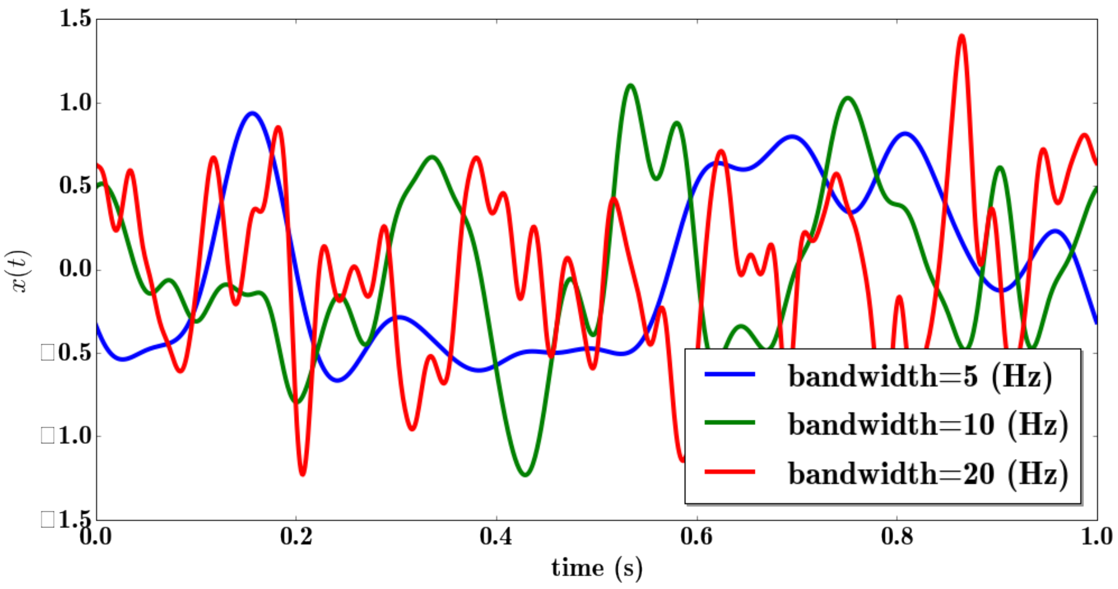


1.2 Gaussian power spectrum noise

Create a modified version of your function from question 1.1 that produces noise with a different power spectrum. Instead of having the $X(\omega)$ values be 0 outside of some limit and sampled from $N(\mu = 0, \sigma = 1)$ inside that limit, we want a smooth drop-off of power as the frequency increases. In particular, instead of the `limit`, we sample from $N(\mu = 0, \sigma = e^{-\omega^2/(2*b^2)})$ where b is the new `bandwidth` parameter that replaces the `limit` parameter.

1.2.1 Plot $x(t)$ for three randomly generated signals with `bandwidth` at 5, 10, and 20Hz. For each of these, `T`=1, `dt`=0.001, and `rms`=0.5.

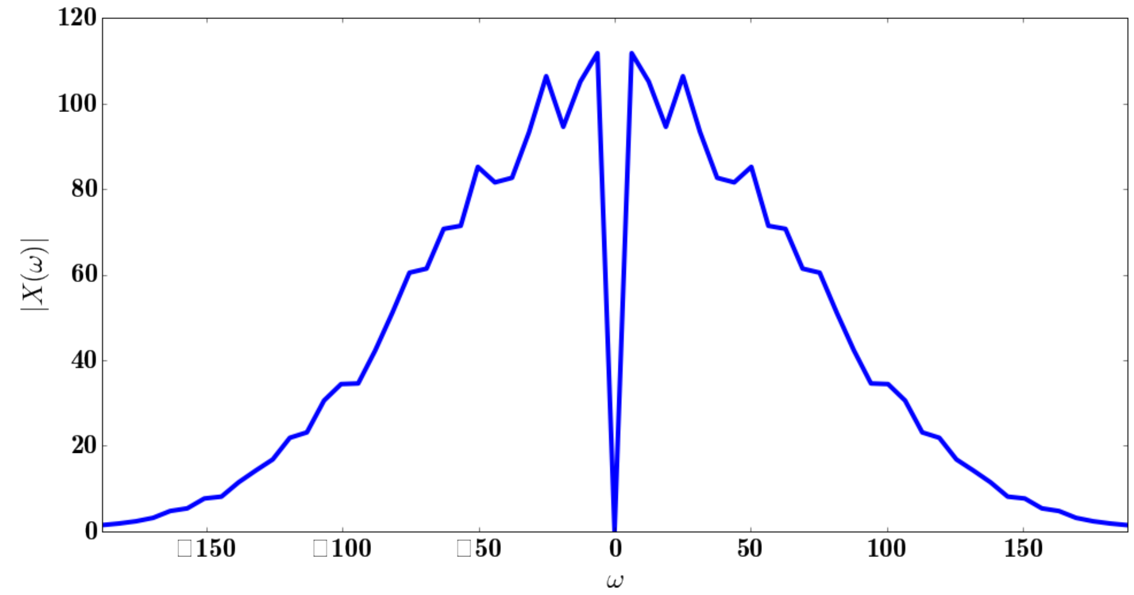
```
In [122]: def one_pt_two_a():
        T=1
        dt=0.001
        rms=0.5
        seed=1
        t=np.arange(int(T/dt)+1)*dt

        bandwidths=[5,10,20]
        x_t_list=[]
        for i in range(len(bandwidths)):
            seed=i
            bandwidth=bandwidths[i]
            x_ti, x_wi = generate_signal(T,dt,rms,bandwidth,seed,'gaussian')
            x_t_list.append(x_ti)

        fig=plt.figure(figsize=(16,8))
        ax=fig.add_subplot(111)
        for i in range(len(bandwidths)):
            ax.plot(t,x_t_list[i],label='bandwidth=%s (Hz)' %(bandwidths[i]))
        ax.set_xlabel('time (s)')
        ax.set_ylabel('$x(t)$')
        legend=ax.legend(loc='best',shadow=True)
        plt.show()

    one_pt_two_a()
```



1.2.2 Plot the average $|X(\omega)|$ (the norm of the Fourier coefficients) over 100 signals generated with T=1, dt=0.001, rms=0.5, and bandwidth=10 (each of these 100 signals should have a different seed)

```
In [123]: def one_pt_two_b():

              T=1
```

```
    dt=0.001
    rms=0.5
    bandwidth=10
    avgs=100

    x_w_list=[]
    for i in range(avgs):
        seed=i
        x_ti, x_wi = generate_signal(T,dt,rms,bandwidth,seed,'gaussian')
        x_w_list.append(np.abs(x_wi))
    x_w_avg=np.average(x_w_list,axis=0)
    w_vals=np.fft.fftfreq(len(x_w_avg))*2*np.pi/dt
    w_limit=2*np.pi*bandwidth

    fig=plt.figure(figsize=(16,8))
    ax=fig.add_subplot(111)
    ax.plot(np.sort(w_vals),np.fft.fftshift(x_w_avg))
    ax.set_xlabel('$\omega$')
    ax.set_ylabel('$|X(\omega)|$')
    ax.set_xlim(-w_limit*3, w_limit*3)
    plt.show()

one_pt_two_b()
```



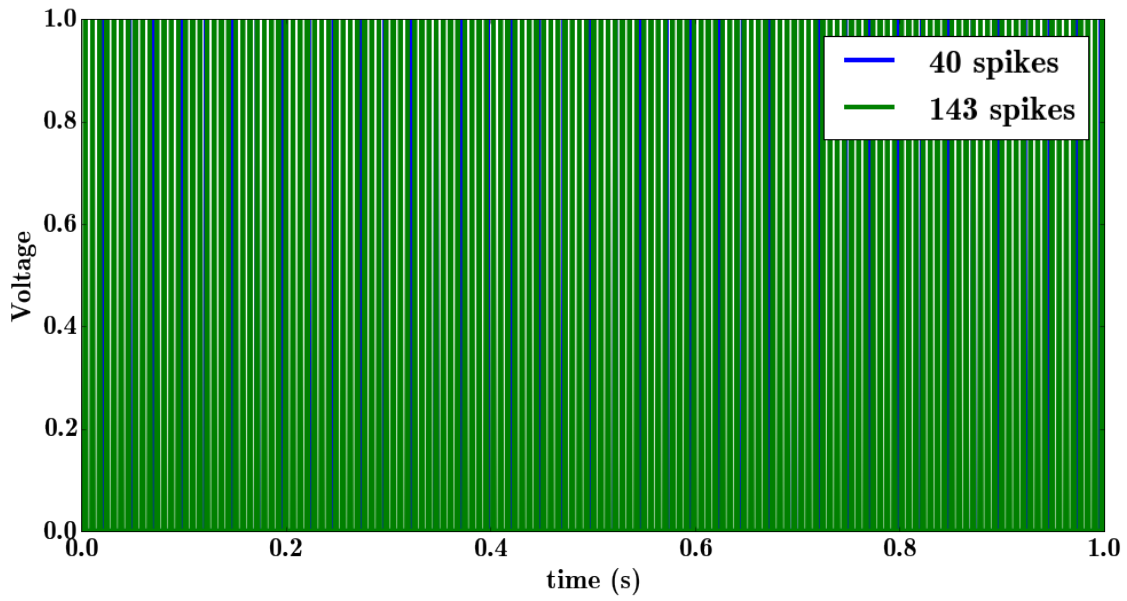
2 Simulating a Spiking Neuron

Write a program to simulate a single Leaky-Integrate and Fire neuron. The core equation is $\frac{dV}{dT} = \frac{1}{\tau_{RC}}(J-V)$ (to simplify life, this is normalized so that R=1, the resting voltage is 0 and the firing voltage is 1). This equation can be simulated numerically by taking small time steps (Euler's method). When the voltage reaches the threshold 1, the neuron will spike and then reset its voltage to 0 for the next τ_{ref} amount of time. Also, if the voltage goes below zero at any time, reset it back to zero. For this question, τ_{RC}=0.02

and τ_{ref}=0.002 Since we want to do inputs in terms of x, we need to do $J = \alpha e \cdot x + J^{bias}$. For this neuron, set e to +1 and find α and J^{bias} such that the firing rate when $x = 0$ is 40Hz and when $x = 1$ it is 150Hz. To find these α and J^{bias} values, use the approximation for the LIF neuron $a(J) = \frac{1}{\tau_{ref} - \tau_{RC} ln(1-\frac{1}{J})}$.

2.1 Plot the spike output for a constant input of $x = 0$ over 1 second. Report the number of spikes. Do the same thing for $x = 1$. Use dt=0.001 for the simulation.

```
In [124]: def two_a():

              x1=0
              x2=1
              a1=40
              a2=150
              encoder=1
              tau_ref=0.002
              tau_rc=0.02
              T=1.0
              dt=0.001

              n1=spikingLIFneuron(x1,x2,a1,a2,encoder,tau_ref,tau_rc)  #create spiking neurons
              stimulus1 = np.linspace(0.0,0.0,T/dt)  #constant stimulus of zero
              n1.set_spikes(stimulus1,T,dt)  #set the spikes of the neuron, given the stimulus
              spikes1=n1.get_spikes()
              stimulus2 = np.linspace(1.0,1.0,T/dt)  #constant stimulus of one
              n1.set_spikes(stimulus2,T,dt)
              spikes2=n1.get_spikes()

              fig=plt.figure(figsize=(16,8))
              ax=fig.add_subplot(111)
              times=np.arange(0,T,dt)
              ax.plot(times,spikes1, 'b', label='%s spikes' %np.count_nonzero(spikes1))
              ax.plot(times,spikes2, 'g', label='%s spikes' %np.count_nonzero(spikes2))
              ax.set_xlabel('time (s)')
              ax.set_ylabel('Voltage')
              legend=ax.legend(loc='best')
              plt.show()

          two_a()
```



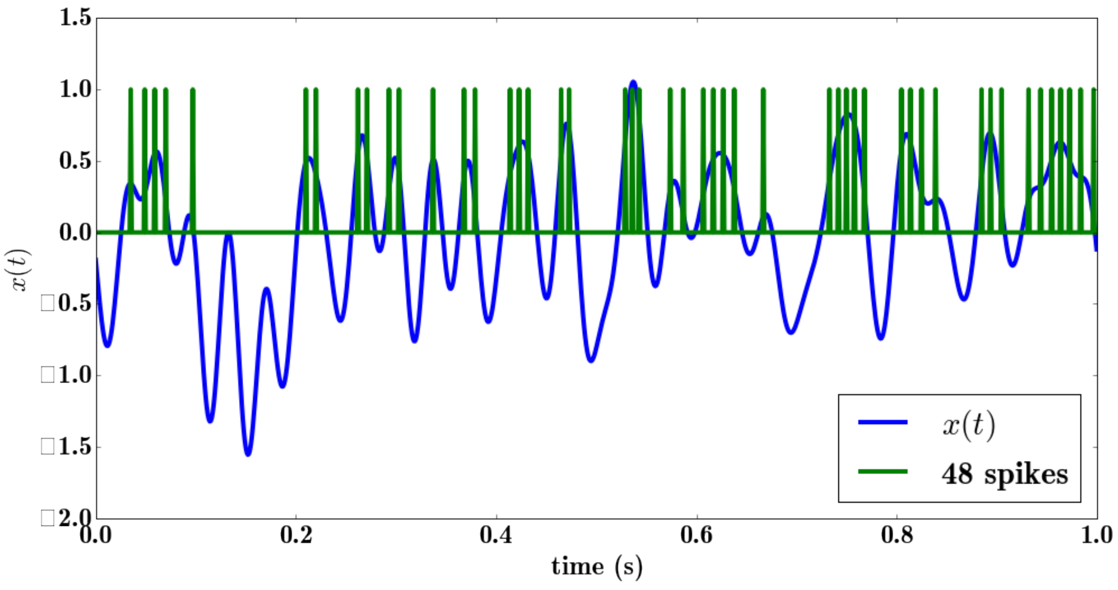
2.2 Does the observed number of spikes in the previous part match the expected number of spikes for $x = 0$ and $x = 1$? Why or why not? What aspects of the simulation would affect this accuracy?

The number of spikes we expect at $stimulus = 0$ is $40Hz$, because this is the firing rate corresponding to $x(t) = 0$ (no external stimulus) that we originally specified. When $stimulus = 1$, we expect the firing rate corresponding to $x(t) = 1$, which is $143Hz$ when we specified a rate of $150hz$. Decreasing the timestep dt improves this accuracy (goes to 149 with $dt = 0.0001$), because Euler's method becomes more exact and because the refractory period is specified with more precision.

2.3 Plot the spike output for $x(t)$ generated using your function from part 1.1. Use T=1, dt=0.001, rms=0.5, and limit=30. Overlay on this plot $x(t)$.

```
In [125]: def two_c():

              rms=0.5
              limit=30
              seed=3
              x1=0
              x2=1
              a1=40
              a2=150
              encoder=1
              tau_ref=0.002
              tau_rc=0.02
              T=1.0
              dt=0.001

              n1=spikingLIFneuron(x1,x2,a1,a2,encoder,tau_ref,tau_rc)
              t=np.arange(int(T/dt)+1)*dt
              x_t, x_w = generate_signal(T,dt,rms,limit,seed,'uniform')
```

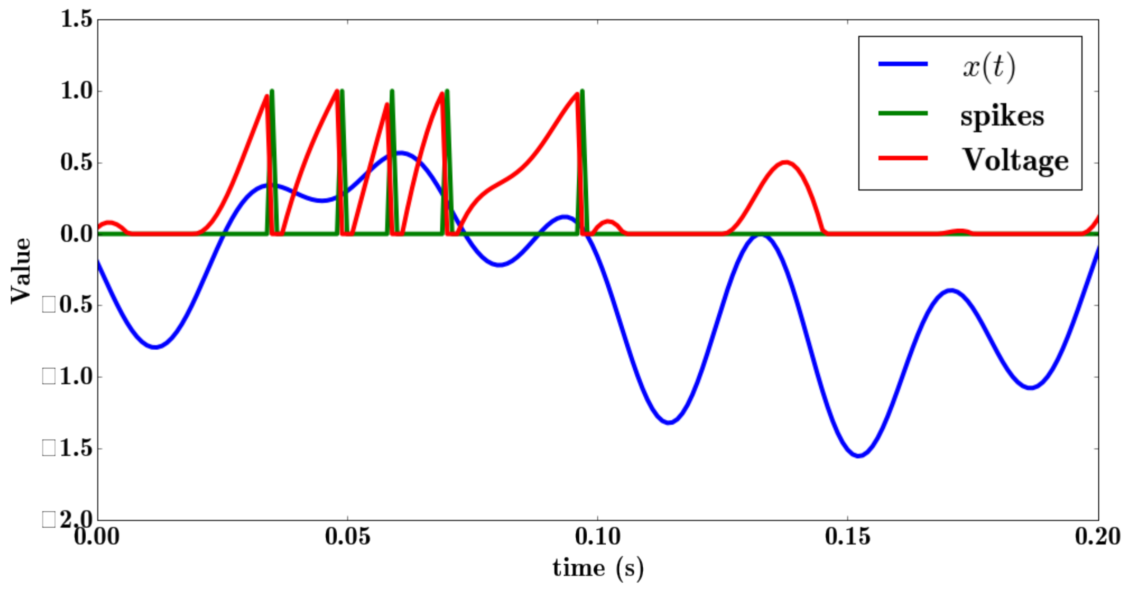
```
        stimulus3 = np.array(x_t)
        n1.set_spikes(stimulus3,T,dt)
        spikes3=n1.get_spikes()

        fig=plt.figure(figsize=(16,8))
        ax=fig.add_subplot(111)
        times=np.arange(0,T,dt)
        ax.plot(t,x_t, label='$x(t)$')
        ax.plot(t,spikes3, label='%s spikes' %np.count_nonzero(spikes3))
        ax.set_xlabel('time (s)')
        ax.set_ylabel('$x(t)$')
        legend=ax.legend(loc='best')
        plt.show()

    two_c()
```



2.4 Using the same $x(t)$ signal as in part (c), plot the neuron's voltage over time for the first 0.2 seconds, along with the spikes over the same time.

```
In [126]: def two_d():

              rms=0.5
              limit=30
              seed=3  #same seed as in part (c)
              x1=0
              x2=1
              a1=40
              a2=150
              encoder=1
              tau_ref=0.002
              tau_rc=0.02
              T=1.0
```

```
    dt=0.001

    n1=spikingLIFneuron(x1,x2,a1,a2,encoder,tau_ref,tau_rc)
    t=np.arange(int(T/dt)+1)*dt
    x_t, x_w = generate_signal(T,dt,rms,limit,seed,'uniform')
    stimulus3 = np.array(x_t)
    n1.set_spikes(stimulus3,T,dt)
    spikes3=n1.get_spikes()

    fig=plt.figure(figsize=(16,8))
    ax=fig.add_subplot(111)
    times=np.arange(0,T,dt)
    ax.plot(t,x_t, label='$x(t)$')
    ax.plot(t,spikes3, label='spikes')
    ax.plot(t,n1.Vhistory, label='Voltage')
    ax.set_xlabel('time (s)')
    ax.set_ylabel('Value')
    ax.set_xlim(0,0.2)
    legend=ax.legend(loc='best')
    plt.show()

two_d()
```



2.5 BONUS: How could you improve this simulation (in terms of how closely the model matches actual equation) without significantly increasing the computation time? 0.5 marks for having a good idea, and up to 1 marks for actually implementing it and showing that it works.

First, you could increase the accuracy of the spike timing by implementing a better ODE solver for voltage in the LIF equation. Rather than using Euler's method to calculate $V(t)$ based on the first order approximation dV/dt, you could use a dedicated solved like 4th/5th order Runge-Kutta, which is the standard for libraries like scipy.integrate. Similarly, you could implement an adaptive stepsize, to capture rapid changes in $x(t)$ without making dt smaller for the entire T and significantly increases computation time.

Second, you could use SVD to calculate the direction(s) of greatest variation within $x(t)$, then set the encoders along this direction (preferably at least one positive and negative). This would ensure that the spike rate of this neuron is highly correlated with changes in the signal, though it wouldn't have much effect in the 1D case.

Third, you could use a different basis function for the neuron tuning curves which better matched the statistics of the signal. Because this signal is derived from a fourier transformation, you could use a basis of sines and cosines with different natural frequencies and amplitudes, as in Georgeopolous or the notes from "Lecture 6 - Decoder Analysis".

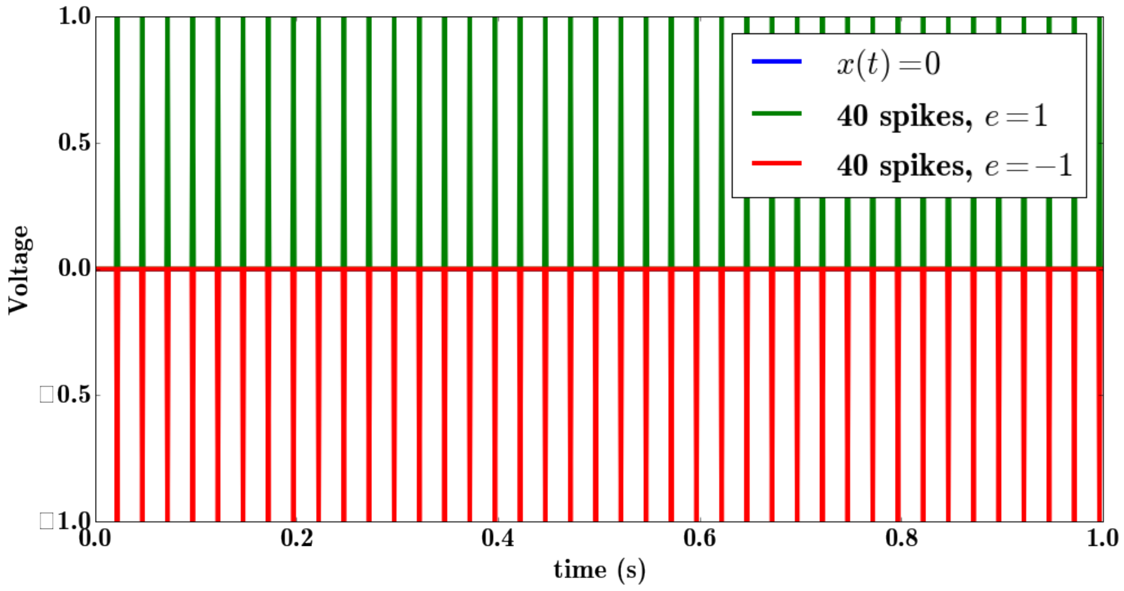
3 Simulating Two Spiking Neurons

Write a program that simulates two neurons. The two neurons have exactly the same parameters, except for one of them $e = 1$ and for the other $e = -1$. Other than that, use exactly the same settings as in question 2.

3.1 Plot $x(t)$ and the spiking output for $x(t) = 0$ (both neurons should spike at ~40 spikes per second)

```
In [127]: def three_a():

              x1=0
              x2=1
              a1=40
              a2=150
              e1=1
              e2=-1
              tau_ref=0.002
              tau_rc=0.02
              T=1
              dt=0.001
              rms=0.5
              limit=30
              seed=3

              #I calculate alpha and Jbias when I initialize the neurons.
              #To do so, I pass the activity a_i corresponding to x_i dot e
              #The "off" neuron with e=-1 should increase its firing rate as a_i
              #x_i decreases, so I pass in x_1=-1 => x_1 dot e = 1, a_1=150
              x1_dot_e1=np.dot(x1,e1)
              x2_dot_e1=np.dot(x2,e1)
              x1_dot_e2=np.dot(x1,e2)
              x2_dot_e2=np.dot(-x2,e2)
              n1=spikingLIFneuron(x1_dot_e1,x2_dot_e1,a1,a2,e1,tau_ref,tau_rc)
              n2=spikingLIFneuron(x1_dot_e2,x2_dot_e2,a1,a2,e2,tau_ref,tau_rc)
              t=np.arange(int(T/dt)+1)*dt
              stimulus1 = np.linspace(0,0,T/dt+1)  #constant stimulus of zero
              n1.set_spikes(stimulus1,T,dt)
              n2.set_spikes(stimulus1,T,dt)
              spikes1=n1.get_spikes()
              spikes2=n2.get_spikes()

              fig=plt.figure(figsize=(16,8))
```

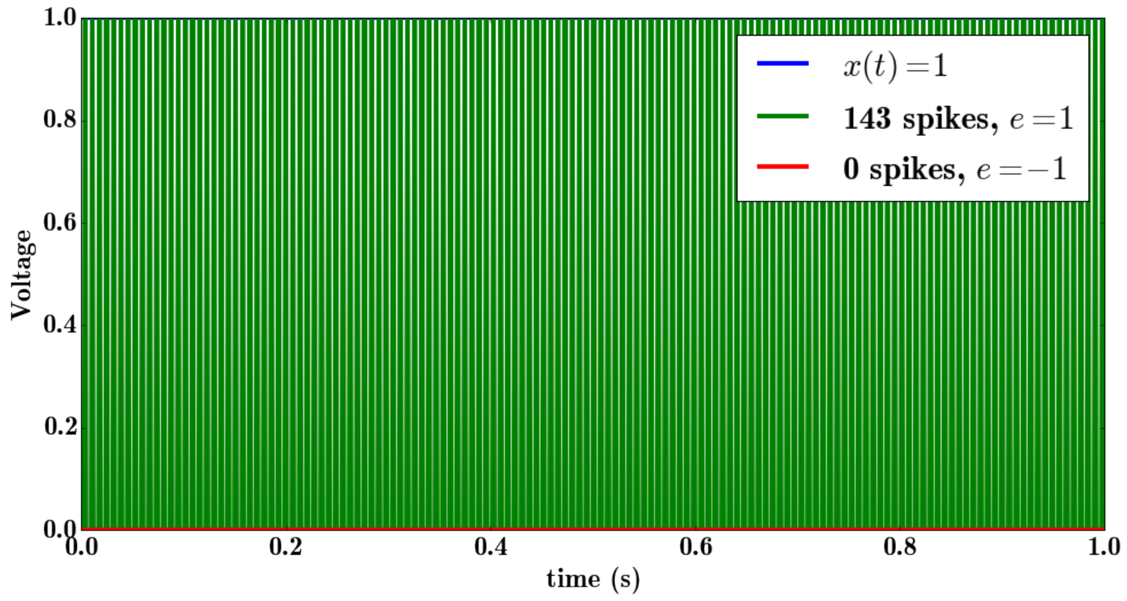
```
        ax=fig.add_subplot(111)
        ax.plot(t,stimulus1, label='$x(t)=0$')
        ax.plot(t,spikes1, label='%s spikes, $e=1$' %np.count_nonzero(spikes1))
        #"off" neuron spikes go in opposite direction
        ax.plot(t,-np.array(spikes2), label='%s spikes, $e=-1$'
                %np.count_nonzero(spikes2))
        ax.set_xlabel('time (s)')
        ax.set_ylabel('Voltage')
        ax.set_xlim(0,T)
        legend=ax.legend(loc='best')
        plt.show()

    three_a()
```



3.2 Plot $x(t)$ and the spiking output for $x(t) = 1$ (one neuron should spike at ˜150 spikes per second, and the other should not spike at all)

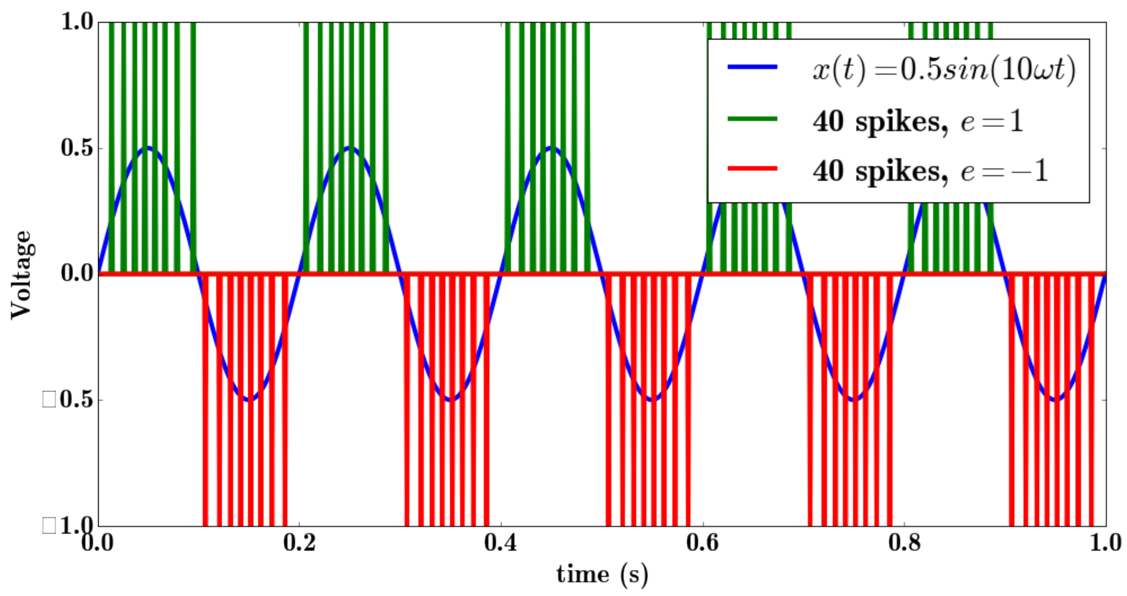
```
In [128]: def three_b():

              x1=0
              x2=1
              a1=40
              a2=150
              e1=1
              e2=-1
              tau_ref=0.002
              tau_rc=0.02
              T=1
              dt=0.001
              rms=0.5
              limit=30
              seed=3
```

```
    x1_dot_e1=np.dot(x1,e1)
    x2_dot_e1=np.dot(x2,e1)
    x1_dot_e2=np.dot(x1,e2)
    x2_dot_e2=np.dot(-x2,e2)
    n1=spikingLIFneuron(x1_dot_e1,x2_dot_e1,a1,a2,e1,tau_ref,tau_rc)
    n2=spikingLIFneuron(x1_dot_e2,x2_dot_e2,a1,a2,e2,tau_ref,tau_rc)
    t=np.arange(int(T/dt)+1)*dt
    stimulus2 = np.linspace(1,1,T/dt+1)
    n1.set_spikes(stimulus2,T,dt)
    n2.set_spikes(stimulus2,T,dt)
    spikes1=n1.get_spikes()
    spikes2=n2.get_spikes()

    fig=plt.figure(figsize=(16,8))
    ax=fig.add_subplot(111)
    ax.plot(t,stimulus2, label='$x(t)=1$')
    ax.plot(t,spikes1, label='%s spikes, $e=1$' %np.count_nonzero(spikes1))
    ax.plot(t,-np.array(spikes2), label='%s spikes, $e=-1$'
            %np.count_nonzero(spikes2))
    ax.set_xlabel('time (s)')
    ax.set_ylabel('Voltage')
    ax.set_xlim(0,T)
    # ax.set_ylim(0,2)
    legend=ax.legend(loc='best')
    plt.show()

three_b()
```



3.3 Plot $x(t)$ and the spiking output for $x(t) = \frac{1}{2}sin(10\pi t)$ (a sine wave at 5Hz).

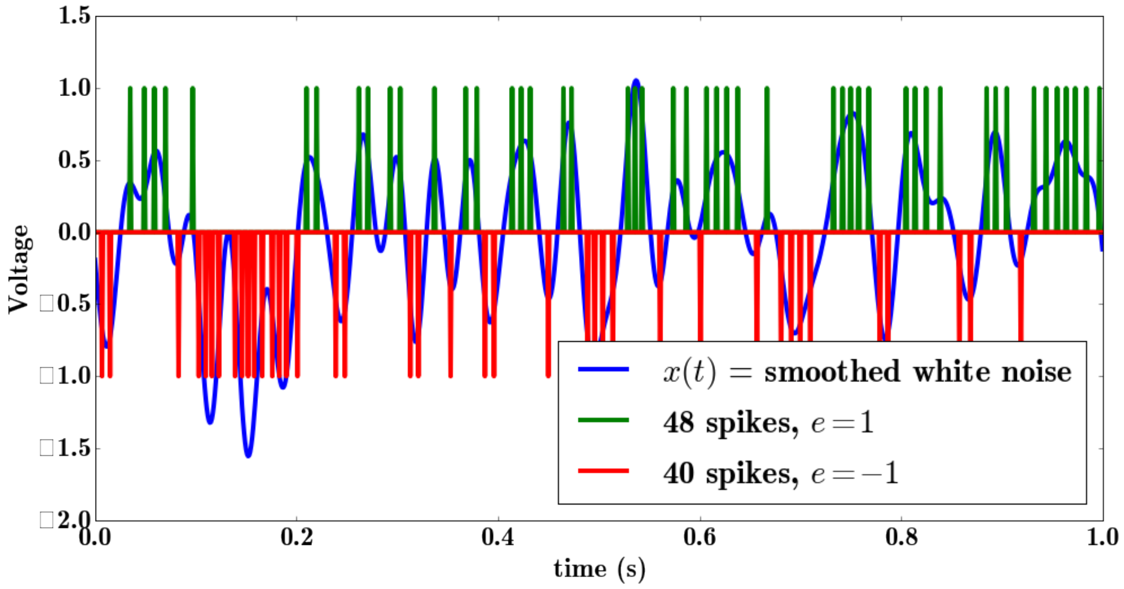
```
In [129]: def three_c():

              x1=0
              x2=1
              a1=40
              a2=150
              e1=1
              e2=-1
              tau_ref=0.002
              tau_rc=0.02
              T=1
              dt=0.001
              rms=0.5
              limit=30
              seed=3

              x1_dot_e1=np.dot(x1,e1)
              x2_dot_e1=np.dot(x2,e1)
              x1_dot_e2=np.dot(x1,e2)
              x2_dot_e2=np.dot(-x2,e2)
              n1=spikingLIFneuron(x1_dot_e1,x2_dot_e1,a1,a2,e1,tau_ref,tau_rc)
              n2=spikingLIFneuron(x1_dot_e2,x2_dot_e2,a1,a2,e2,tau_ref,tau_rc)
              t=np.arange(int(T/dt)+1)*dt
              stimulus3 = 0.5*np.sin(10*np.pi*t)
              n1.set_spikes(stimulus3,T,dt)
              n2.set_spikes(stimulus3,T,dt)
              spikes1=n1.get_spikes()
              spikes2=n2.get_spikes()

              fig=plt.figure(figsize=(16,8))
              ax=fig.add_subplot(111)
              ax.plot(t,stimulus3, label='$x(t)=0.5 sin(10 \omega t)$')
              ax.plot(t,spikes1, label='%s spikes, $e=1$' %np.count_nonzero(spikes1))
              ax.plot(t,-np.array(spikes2), label='%s spikes, $e=-1$'
                      %np.count_nonzero(spikes2))
              ax.set_xlabel('time (s)')
              ax.set_ylabel('Voltage')
              ax.set_xlim(0,T)
              legend=ax.legend(loc='best')
              plt.show()

          three_c()
```



3.4 Plot $x(t)$ and the spiking output for a random signal generated with your function for question 1.1 with T=2, dt=0.001, rms=0.5, and limit=5.

```
In [130]: def three_d():

              x1=0
              x2=1
              a1=40
              a2=150
              e1=1
              e2=-1
              tau_ref=0.002
              tau_rc=0.02
              T=1
              dt=0.001
              rms=0.5
              limit=30
              seed=3

              x1_dot_e1=np.dot(x1,e1)
              x2_dot_e1=np.dot(x2,e1)
              x1_dot_e2=np.dot(x1,e2)
              x2_dot_e2=np.dot(-x2,e2)
              n1=spikingLIFneuron(x1_dot_e1,x2_dot_e1,a1,a2,e1,tau_ref,tau_rc)
              n2=spikingLIFneuron(x1_dot_e2,x2_dot_e2,a1,a2,e2,tau_ref,tau_rc)
              t=np.arange(int(T/dt)+1)*dt
              x_t, x_w = generate_signal(T,dt,rms,limit,seed,'uniform')
              stimulus4 = np.array(x_t)
              n1.set_spikes(stimulus4,T,dt)
              n2.set_spikes(stimulus4,T,dt)
              spikes1=n1.get_spikes()
              spikes2=n2.get_spikes()
```

```
    fig=plt.figure(figsize=(16,8))
    ax=fig.add_subplot(111)
    ax.plot(t,stimulus4, label='$x(t)$ = smoothed white noise')
    ax.plot(t,spikes1, label='%s spikes, $e=1$' %np.count_nonzero(spikes1))
    ax.plot(t,-np.array(spikes2), label='%s spikes, $e=-1$'
            %np.count_nonzero(spikes2))
    ax.set_xlabel('time (s)')
    ax.set_ylabel('Voltage')
    ax.set_xlim(0,T)
    legend=ax.legend(loc='best')
    plt.show()

three_d()
```



4 Computing an Optimal Filter

Compute the optimal filter for decoding pairs of spikes. Instead of implementing this yourself, here is an implementation in Python and an implementation in Matlab.

4.1 **Document the code and connect it with the code you wrote for part (3) so that it uses the signal used in 3.d. Comments should be filled in where there are # signs (Python) or % signs (Matlab). Replace the '???' labels in the code with the correct labels.**

4.2 **Plot the time and frequency plots for the optimal filter for the signal you generated in question 3.d.**

4.3 **Plot the $x(t)$ signal, the spikes, and the decoded $\hat{x}(t)$ value for the signal in question 3.d.**

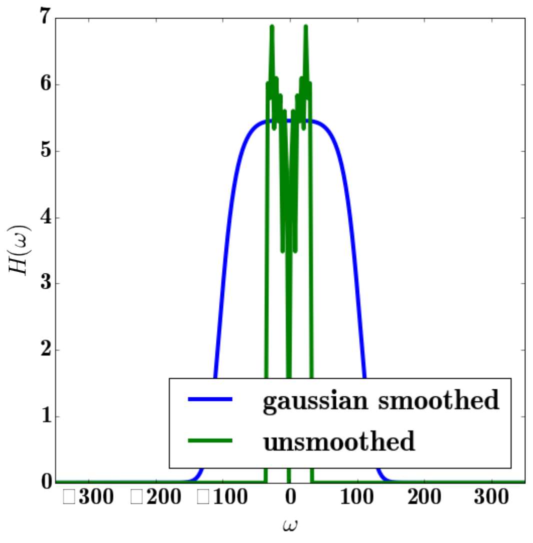
4.4 **Plot the $|X(\omega)|$ power spectrum, $|R(\omega)|$ spike response spectrum, and the $|\hat{X}(\omega)|$ power spectrum for the signal in question 3.d. How do these relate to the optimal filter?**

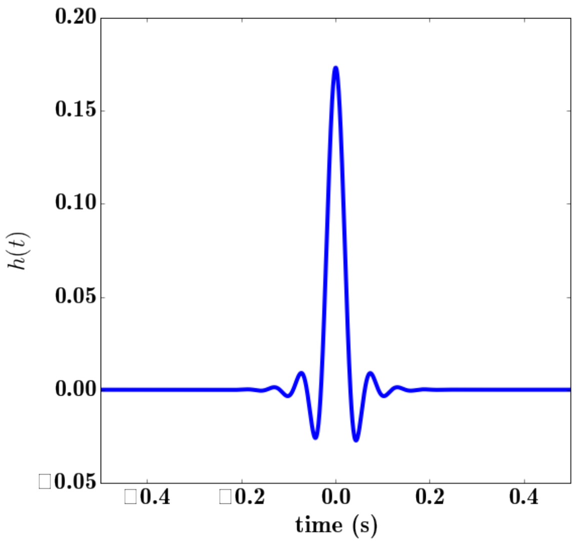
```
In [131]: def four_a_thru_d():

              T = 2.0          # length of signal in seconds
              dt = 0.001       # time step size
              rms=0.5
              limit=5
              seed=3

              # Generate bandlimited white noise (use your own function from part 1.1)
              x_t, x_w = generate_signal(T,dt,rms,limit,seed,'uniform')

              Nt = len(x_t)              #length of the returned signal
              t = np.arange(Nt) * dt     #time values of the signal

              # Neuron parameters
              tau_ref = 0.002             # refractory period in seconds
              tau_rc = 0.02               # RC time constant in seconds
              x1 = 0.0                    # firing rate at x=x0 is a0
              a1 = 40.0
              x2 = 1.0                    # firing rate at x=x1 is a1
              a2 = 150.0

              #I actually calculate alpha and Jbias when I initialize the neurons,
              #so I'm leaving out your code
              e1=1
              e2=-1
              x1_dot_e1=np.dot(x1,e1)
              x2_dot_e1=np.dot(x2,e1)
              x1_dot_e2=np.dot(x1,e2)
              x2_dot_e2=np.dot(-x2,e2)
              #create spiking neurons
              n1=spikingLIFneuron(x1_dot_e1,x2_dot_e1,a1,a2,e1,tau_ref,tau_rc)
              n2=spikingLIFneuron(x1_dot_e2,x2_dot_e2,a1,a2,e2,tau_ref,tau_rc)
              #use the generated white-noise signal as stimulus
              stimulus4 = np.array(x_t)
              #calculate the spikes corresponding to this signal
              n1.set_spikes(stimulus4,T,dt)
              n2.set_spikes(stimulus4,T,dt)
              #return the spikes
```

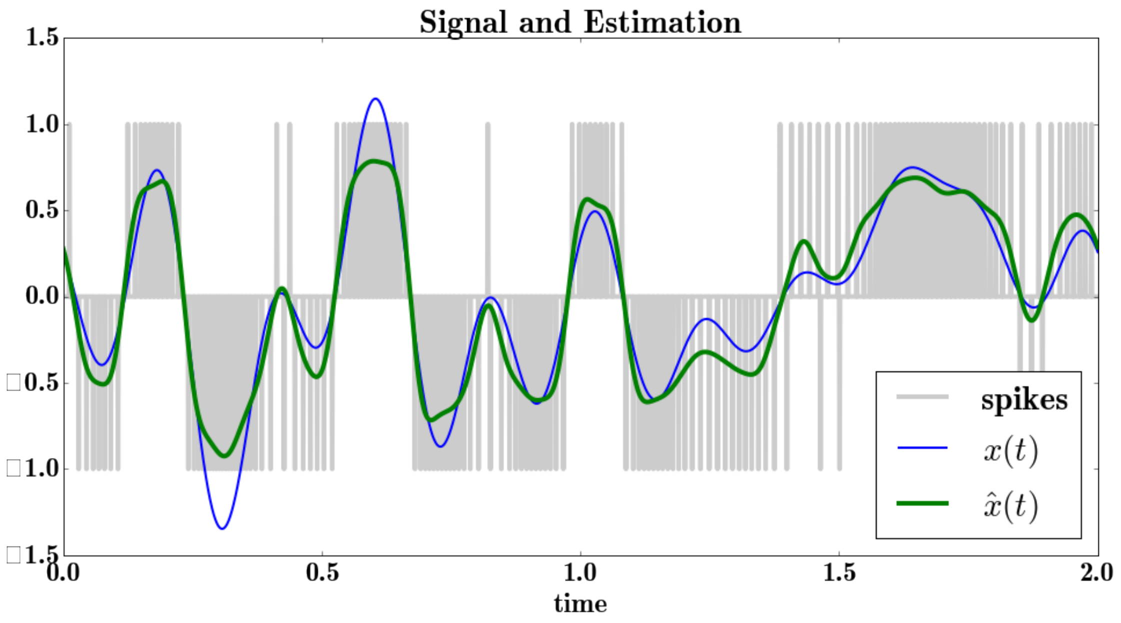
```
spikes1=n1.get_spikes()
spikes2=n2.get_spikes()
#put spikes in the given form
spikes=np.array([spikes1,spikes2])

#frequencies in Hz of the signal (-f_max to +f_max)
freq = np.arange(Nt)/T - Nt/(2.0*T)
#corresponding frequencies in radians (-w_max to w_max)
omega = freq*2*np.pi

# the response of the two neurons combined together:
#nonzero values at each time step when one neuron spiked and the other did not
r = spikes[0] - spikes[1]
# translate this response difference into the frequency domain,
# this will turn convolution into multiplication and represents spike train power
R = np.fft.fftshift(np.fft.fft(r))
# X is frequency domain description of the white noise signal
X=np.fft.fftshift(x_w)

#width of the gaussian window filter in frequency space
#sigma_t = 1/tau, so with smaller sigma creates larger temporal windows
sigma_t = 0.025
#Gaussian filter, expressing power as a function of the freqency omega
W2 = np.exp(-omega**2*sigma_t**2)
#Normalize the filter from 0 to 1
W2 = W2 / sum(W2)

# X(w)*R*(w) represents correlated power between signal and spike train amplitudes
CP = X*R.conjugate()
# convolve correlated power with the gaussian to perform windowed filtering
WCP = np.convolve(CP, W2, 'same')
# calculate spike train power |R(w_n:A)|^2 =  R * complex conjugate R
RP = R*R.conjugate()
# convolve spike train power with the gaussian to perform windowed filtering
WRP = np.convolve(RP, W2, 'same')
# calculate the signal power spectrum |X(w)|^2 = X * complex congugate X
XP = X*X.conjugate()
#convolve the R(w) values for the neuron pair response with the gaussian filter
#not used here, but it could be used to calculate the error in frequency domain
WXP = np.convolve(XP, W2, 'same')
#The optimal temporal filter in the frequency domain; equation 4.19 in NEF book
H = WCP / WRP
H_unsmoothed = CP / RP #for comparison

# bring the optimal temporal filter into the time domain,
#making sure the frequencies line up properly
h = np.fft.fftshift(np.fft.ifft(np.fft.ifftshift(H))).real

#convolve temporal filter and neuron pair response in the frequency domain,
#returns the temporally filtered estimate for spiking neurons
XHAT = H*R
#bring the frequency domain state estimate into the time domain
xhat = np.fft.ifft(np.fft.ifftshift(XHAT)).real
```

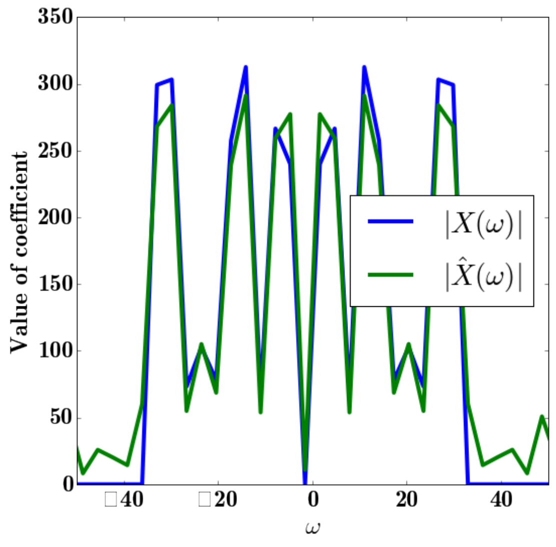
```
#4b
fig=plt.figure(figsize=(16,8))
# plt.title('Optimal Temporal Filter')
ax=fig.add_subplot(121)
#optimal temporal filter power in frequency domain
ax.plot(omega,H.real, label='gaussian smoothed')
ax.plot(omega,H_unsmoothed.real, label='unsmoothed')
ax.set_xlabel('$\omega$')
ax.set_ylabel('$H(\omega)$')
ax.set_xlim(-350,350)
legend=ax.legend(loc='lower right')
ax=fig.add_subplot(122)
ax.plot(t-T/2,h)      #temporal filter power in the time domain
ax.set_xlabel('time (s)')
ax.set_ylabel('$h(t)$')
ax.set_xlim(-0.5,0.5)
plt.tight_layout()
plt.show()

#4c
# plt.title('Time Domain')
fig=plt.figure(figsize=(16,8))
plt.title('Signal and Estimation')
ax=fig.add_subplot(111)
#neuron pair spikes, time domain
plt.plot(t, r, color='k', label='spikes', alpha=0.2)
#white noise signal, time domain
plt.plot(t, x_t, linewidth=2, label='$x(t)$')
#normed estimated state, time domain
plt.plot(t, xhat, label='$\hat{x}(t)$')
legend=ax.legend(loc='best')
ax.set_xlabel('time')
plt.show()

#4c
fig=plt.figure(figsize=(16,8))
# plt.title('Frequency Domain')
# .real removes unnecessary warnings when plotting
ax=fig.add_subplot(121)
#unsmoothed white noise signal in frequency domain
ax.plot(omega,np.sqrt(XP).real, label='$|X(\omega)|$')
#state estimate in frequency domain
ax.plot(omega,np.abs(XHAT).real, label='$|\hat{X}(\omega)|$')
ax.set_xlabel('$\omega$')
ax.set_ylabel('Value of coefficient')
ax.set_xlim(-50,50)
legend=ax.legend(loc='best')
ax=fig.add_subplot(122)
#unsmoothed neuron pair spikes frequency domain
ax.plot(omega,np.sqrt(RP).real, label='$|R(\omega)|$')
ax.set_xlabel('$\omega$')
ax.set_ylabel('Value of coefficient')
legend=ax.legend(loc='best')
legend=ax.legend(loc='best')
```

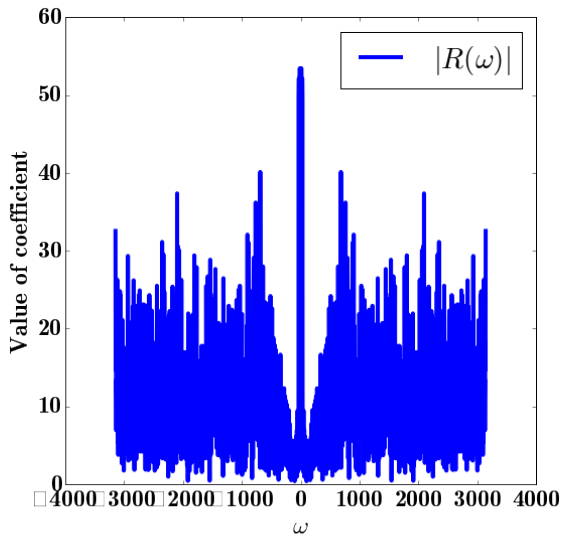
```
        legend=ax.legend(loc='best')
        plt.tight_layout()
        plt.show()

    four_a_thru_d()
```











The power spectrum for X and $\hat{X}$ align fairly closely, giving a first indication that the state estimate is capturing the original signal. $\hat{X}$ does have spurious power above the cutoff due to the smoothing procedures, which ultimately gives the time-domain signal more high-frequency components than it should have.

$\hat{X}$ is calculated by convolving the optimal filter in the time domain $h(t)$ with the neuron response $r(t)$, or by multiplying $H(\omega)$ with $R(\omega)$ in the frequency domain and taking the inverse fourier transform:

- $\hat{X}(\omega) = H(\omega)R(\omega)$.

$X(\omega)$ and $R(\omega)$ are filtered with gaussian windowing in these plots, but this is an arbitrary filter that isn't related to $H(\omega)$. They are, however, used to calculate $H(\omega)$:

- $H(\omega) = \frac{\langle X(\omega)R^*(\omega)\rangle_A}{\langle |R(\omega)|^2\rangle_A}$,

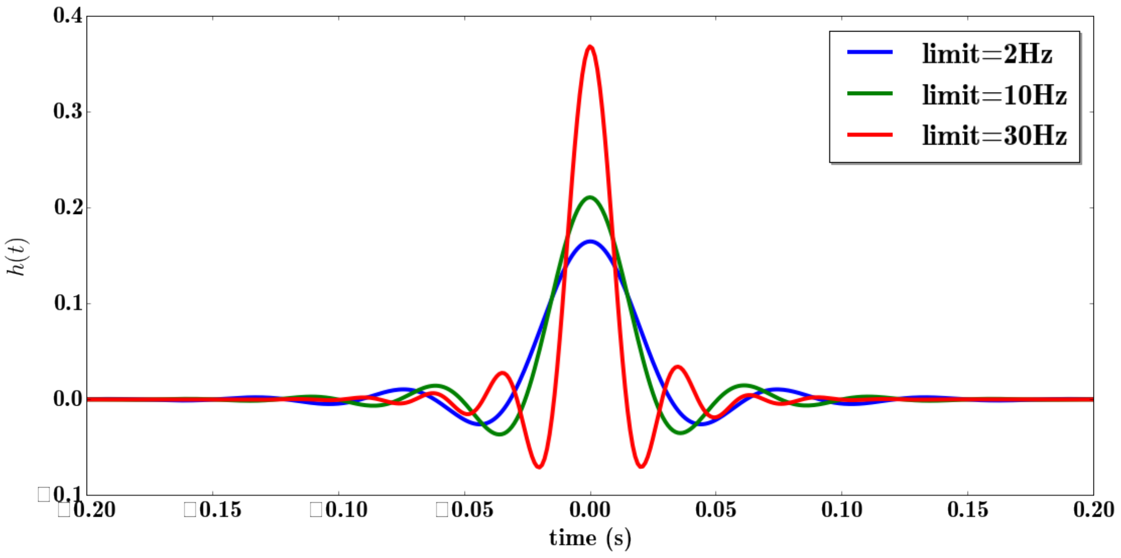
where $\langle\rangle_A$ is the averaging over the gaussian window A.

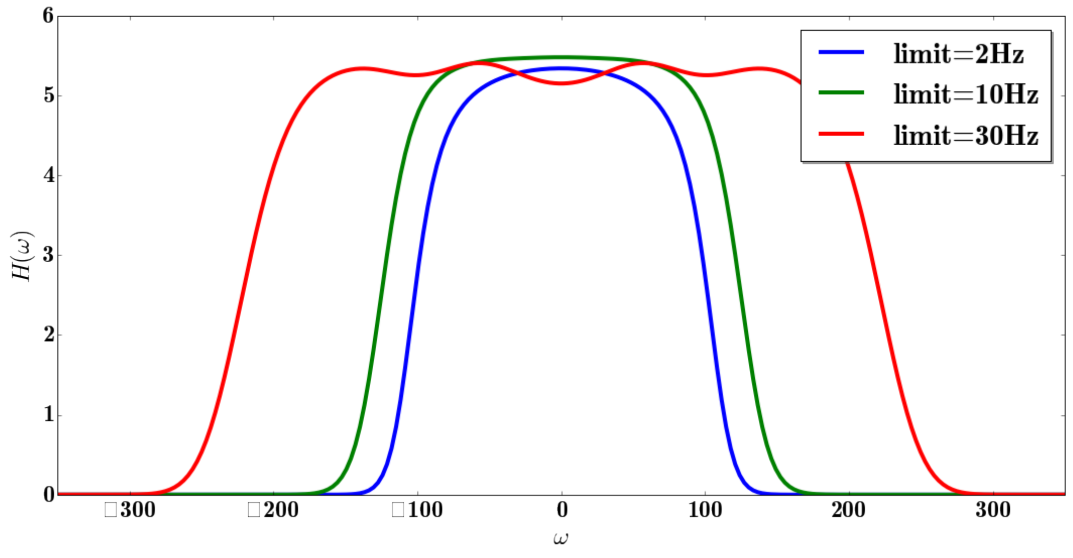
4.5 Generate $h(t)$ time plots for the optimal filter for different `limit` values of 2Hz, 10Hz, and 30Hz. Describe the effects on the time plot of the optimal filter as the `limit` increases. Why does this happen?

```
In [132]: def four_e():

                T = 2.0          # length of signal in seconds
                dt = 0.001       # time step size
                rms=0.5
                limit=5
                seed=3

                # Neuron creation and spike generation
                tau_ref = 0.002
                tau_rc = 0.02
                x1 = 0.0
                a1 = 40.0
                x2 = 1.0
                a2 = 150.0
```

```
e1=1
e2=-1
x1_dot_e1=np.dot(x1,e1)
x2_dot_e1=np.dot(x2,e1)
x1_dot_e2=np.dot(x1,e2)
x2_dot_e2=np.dot(-x2,e2)
n1=spikingLIFneuron(x1_dot_e1,x2_dot_e1,a1,a2,e1,tau_ref,tau_rc)
n2=spikingLIFneuron(x1_dot_e2,x2_dot_e2,a1,a2,e2,tau_ref,tau_rc)

fig=plt.figure(figsize=(16,16))
ax=fig.add_subplot(211)
ax.set_xlabel('time (s)')
ax.set_ylabel('$h(t)$')
ax.set_xlim(-0.2,0.2)
ax2=fig.add_subplot(212)
ax2.set_xlabel('$\omega$')
ax2.set_ylabel('$H(\omega)$')
ax2.set_xlim(-350,350)

limits=[2,10,30]
for i in range(len(limits)):
    seed=i
    limit=limits[i]
    x_t, x_w = generate_signal(T,dt,rms,limit,seed,'uniform')
    Nt = len(x_t)
    t = np.arange(Nt) * dt
    stimulus = np.array(x_t)
    n1.set_spikes(stimulus,T,dt)
    n2.set_spikes(stimulus,T,dt)
    spikes1=n1.get_spikes()
    spikes2=n2.get_spikes()
    spikes=np.array([spikes1,spikes2])

    freq = np.arange(Nt)/T - Nt/(2.0*T)
    omega = freq*2*np.pi

    r = spikes[0] - spikes[1]
    R = np.fft.fftshift(np.fft.fft(r))
    X=np.fft.fftshift(x_w)

    sigma_t = 0.025
    W2 = np.exp(-omega**2*sigma_t**2)
    W2 = W2 / sum(W2)

    CP = X*R.conjugate()
    WCP = np.convolve(CP, W2, 'same')
    RP = R*R.conjugate()
    WRP = np.convolve(RP, W2, 'same')
    XP = X*X.conjugate()
    WXP = np.convolve(XP, W2, 'same')
    H = WCP / WRP
    H_unsmoothed = CP / RP

    h = np.fft.fftshift(np.fft.ifft(np.fft.ifftshift(H))).real
```

```
            ax.plot(t-T/2,h,label='limit=%sHz' %(limits[i]))
            ax2.plot(omega,H.real,label='limit=%sHz' %(limits[i]))

        legend=ax.legend(loc='upper right',shadow=True)
        legend=ax2.legend(loc='upper right',shadow=True)
        plt.tight_layout()
        plt.show()

    four_e()
```





Increasing the frequency limit of the signal compresses the temporal filter in time, causing the decaying oscillations to occur nearer to $t = 0$ and with greater magnitude. Mathematically, increasing the limit increases the width of $H(\omega)$ due to a wider $X(\omega)$ in the numerator $X(\omega) * R(\omega)$. This preserves a greater number of high frequency components and more rapid oscillations in the time domain. These oscillations add coherently near $t = 0$ increasing the filter's power, but add destructively as $t \to \infty$, decreasing the filter's power. Intuitively, having a wider $H(\omega)$ means capturing more high-frequency components. In order to not smooth out these components in the time domain near $t = 0$, the temporal filter $h(t)$ needs to oscillate on short time scales.

5 Using Post-Synaptic Currents as a Filter

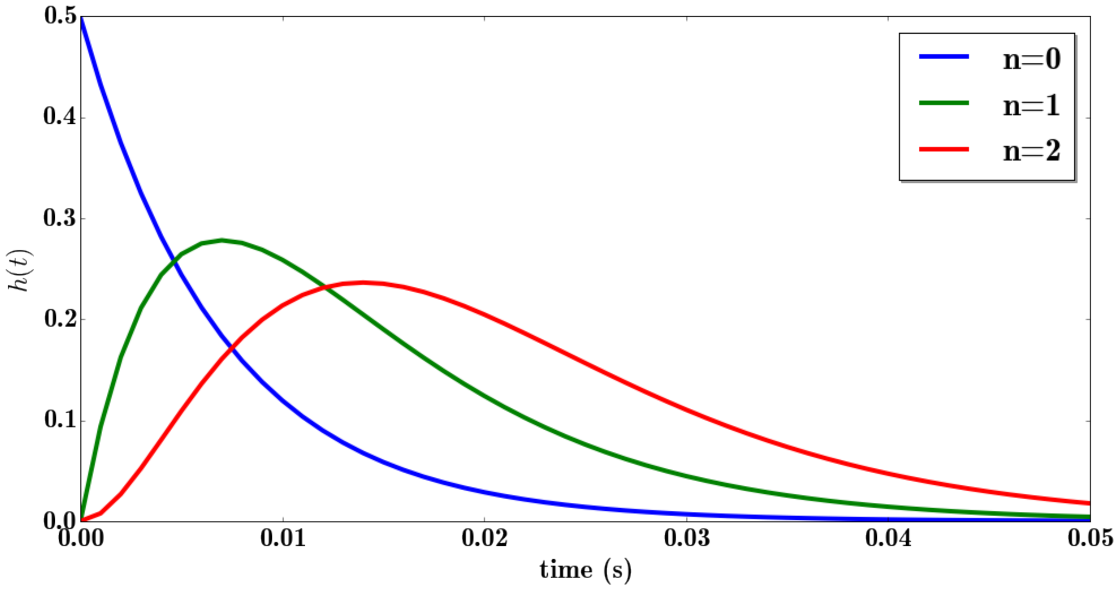
Instead of using the optimal filter from the previous question, now we will use the post-synaptic current instead. This is of the form $h(t) = t^n e^{-t/\tau}$ normalized to area 1.

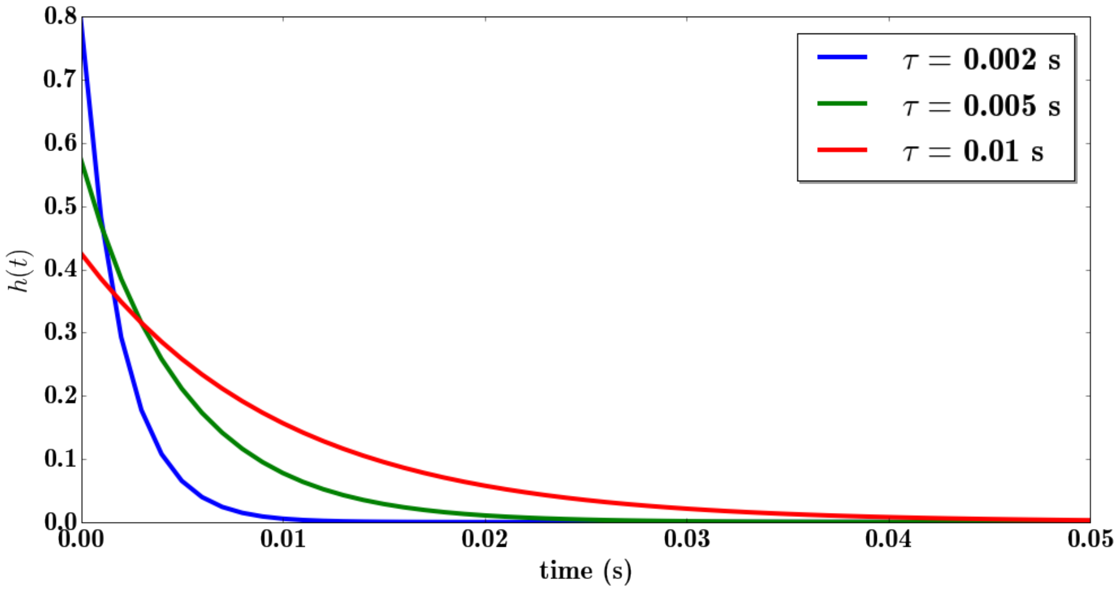
5.1 Plot the normalized $h(t)$ for n=0, 1, and 2 with τ=0.007 seconds. What two things do you expect increasing n will do to $\hat{x}(t)$?

5.2 Plot the normalized $h(t)$ for τ=0.002, 0.005, 0.01, and 0.02 seconds with n=0. What two things do you expect increasing τ will do to $\hat{x}(t)$?

```
In [133]: def five_a_thru_b():

              #5a
              T=1
              dt=0.001
              tau=0.007
              t=np.arange(T/dt)*dt
              n_list=[0,1,2]

              fig=plt.figure(figsize=(16,8))
              ax=fig.add_subplot(111)
              ax.set_xlabel('time (s)')
              ax.set_ylabel('$h(t)$')
              for n in n_list:
                  h=t**n*np.exp(-t/tau)
                  h=h/np.linalg.norm(h)
                  ax.plot(t,h,label='n=%s' %n)
              ax.set_xlim(0,0.05)
              legend=ax.legend(loc='best',shadow=True)
              plt.show()

              #5b
              T=1
              dt=0.001
              n=0
              t=np.arange(T/dt)*dt
              tau_list=[0.002,0.005,0.01]

              fig=plt.figure(figsize=(16,8))
              ax=fig.add_subplot(111)
              ax.set_xlabel('time (s)')
              ax.set_ylabel('$h(t)$')
              for tau in tau_list:
                  h=t**n*np.exp(-t/tau)
                  h=h/np.linalg.norm(h)
                  ax.plot(t,h,label='$\\tau$ = %s s' %tau)
              ax.set_xlim(0,0.05)
              legend=ax.legend(loc='best',shadow=True)
              plt.show()
```

```
five_a_thru_b()
```





I expect increasing n to increase the amount of smoothing on $\hat{x}$, because h extends farther in time with larger n, and so spreads the spikes farther out in time. Given the result of 5a, I also expect that a wider filter will decrease the high-frequency components of the signal. I expect increasing n to cause greater lag in $\hat{x}$, because the peak of h moves farther from zero with larger n (sort of like a shifting center of mass for the filter).

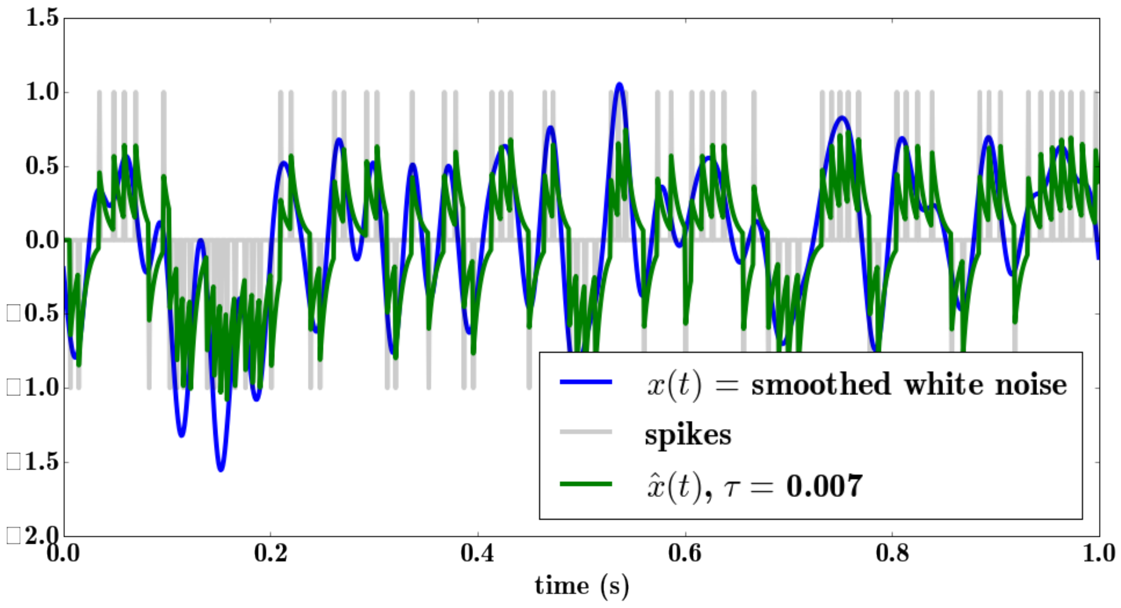
I expect increasing τ to increase the amount of smoothing (in time) on $\hat{x}$, because h extends farther in time with larger τ. This will decrease the magnitude of the decoded signal (due to the lower power of h at each t) as well as reduce high-frequency components, as above.

5.3 **Decode $\hat{x}(t)$ from the spikes generated in question 3.d using an $h(t)$ with n=0 and τ=0.007. Do this by generating the spikes, filtering them with $h(t)$, and using that as your activity matrix A to compute your decoders. Plot the time and frequency plots for this $h(t)$. Plot the $x(t)$ signal, the spikes, and the decoded $\hat{x}(t)$ value.**

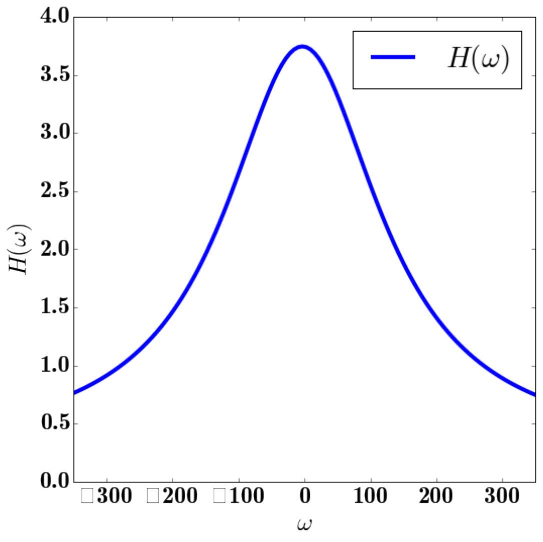
5.4 **Use the same decoder and $h(t)$ as in part (c), but generate a new $x(t)$ with `limit`=5Hz. Plot the $x(t)$ signal, the spikes, and the decoded $\hat{x}(t)$ value. How do these decodings compare?**

```
In [134]: def five_c_thru_d():

              x1=0
              x2=1
              a1=40
              a2=150
              e1=1
              e2=-1
              tau_ref=0.002
              tau_rc=0.02
              T=1
              dt=0.001
              rms=0.5
              limit=30
              n=0
              tau_synapse=0.007
              seed=3      #new seed to generate new x(t)

              #create neurons, generate signals, generate spikes
              x1_dot_e1=np.dot(x1,e1)
              x2_dot_e1=np.dot(x2,e1)
              x1_dot_e2=np.dot(x1,e2)
              x2_dot_e2=np.dot(-x2,e2)
              n1=spikingLIFneuron(x1_dot_e1,x2_dot_e1,a1,a2,e1,tau_ref,tau_rc)
              n2=spikingLIFneuron(x1_dot_e2,x2_dot_e2,a1,a2,e2,tau_ref,tau_rc)
              t=np.arange(int(T/dt)+1)*dt
              x_t, x_w = generate_signal(T,dt,rms,limit,seed,'uniform')
              stimulus = np.array(x_t)
              n1.set_spikes(stimulus,T,dt)
              n2.set_spikes(stimulus,T,dt)
              spikes1=n1.get_spikes()
              spikes2=n2.get_spikes()

              #set post-synaptic current temporal filter
              Nt = len(x_t)
              freq = np.arange(Nt)/T - Nt/(2.0*T)
              omega = freq*2*np.pi
              spikes=np.array([spikes1,spikes2])
              h=t**n*np.exp(-t/tau_synapse)
              h=h/np.linalg.norm(h)
              H=np.fft.fft(h)

              #calculate decoders and filtered state estimate using smoothed spike data
              d=get_decoders_smoothed(spikes,h,x_t)
```

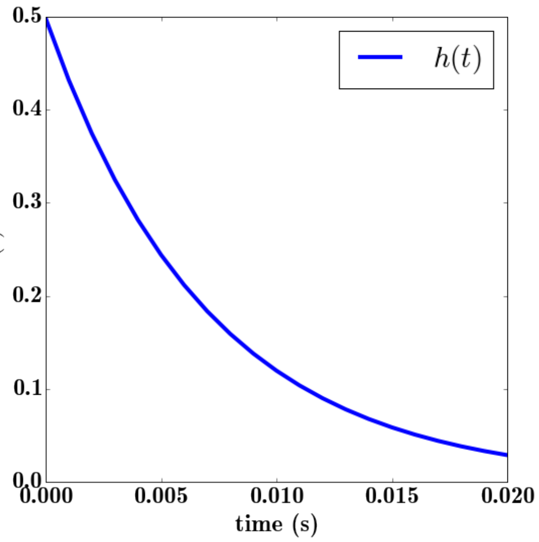
```
    xhat=get_estimate_smoothed(spikes,h,d)

    #plot
    fig=plt.figure(figsize=(16,8))
    ax=fig.add_subplot(111)
    ax.plot(t,x_t, label='$x(t)$ = smoothed white noise')
    #neuron pair response function in time domain
    plt.plot(t,(spikes[0]-spikes[1]), color='k', label='spikes', alpha=0.2)
    ax.plot(t,xhat, label='$\hat{x}(t)$, $\\tau$ = %s' %tau_synapse)
    ax.set_xlabel('time (s)')
    legend=ax.legend(loc='best')
    plt.show()

    fig=plt.figure(figsize=(16,8))
    ax=fig.add_subplot(121)
    ax.plot(omega,np.fft.fftshift(H).real,label='$H(\omega)$')
    ax.set_xlabel('$\omega$')
    ax.set_ylabel('$H(\omega)$')
    ax.set_xlim(-350,350)
    legend=ax.legend(loc='upper right')
    ax=fig.add_subplot(122)
    ax.plot(t,h,label='$h(t)$')
    ax.set_xlabel('time (s)')
    ax.set_ylabel('$h(t)$')
    ax.set_xlim(0,0.02)
    legend=ax.legend(loc='upper right')
    plt.tight_layout()
    plt.show()

    #5d
    seed = 9       #new seed to generate a fresh x(t)
    limit=5        #generate a new signal with limit=5hz
    x_t2, x_w2 = generate_signal(T,dt,rms,limit,seed,'uniform')
    stimulus2 = np.array(x_t2)
    n1.set_spikes(stimulus2,T,dt) #generate new spikes
    n2.set_spikes(stimulus2,T,dt)
    spikes3=n1.get_spikes()
    spikes4=n2.get_spikes()
    spikes2=np.array([spikes3,spikes4])
    xhat2=get_estimate_smoothed(spikes2,h,d) #estimate the new state with the old decoders

    fig=plt.figure(figsize=(16,8))
    ax=fig.add_subplot(111)
    ax.plot(t,x_t2, label='$x(t)$ = smoothed white noise')
    #neuron pair response function in time domain
    plt.plot(t,(spikes2[0]-spikes[1]), color='k', label='spikes', alpha=0.2)
    ax.plot(t,xhat2, label='$\hat{x}(t)$, old decoders, $\\tau$ = %s' %tau_synapse)
    ax.set_xlabel('time (s)')
    legend=ax.legend(loc='best')
    plt.show()

five_c_thru_d()
```



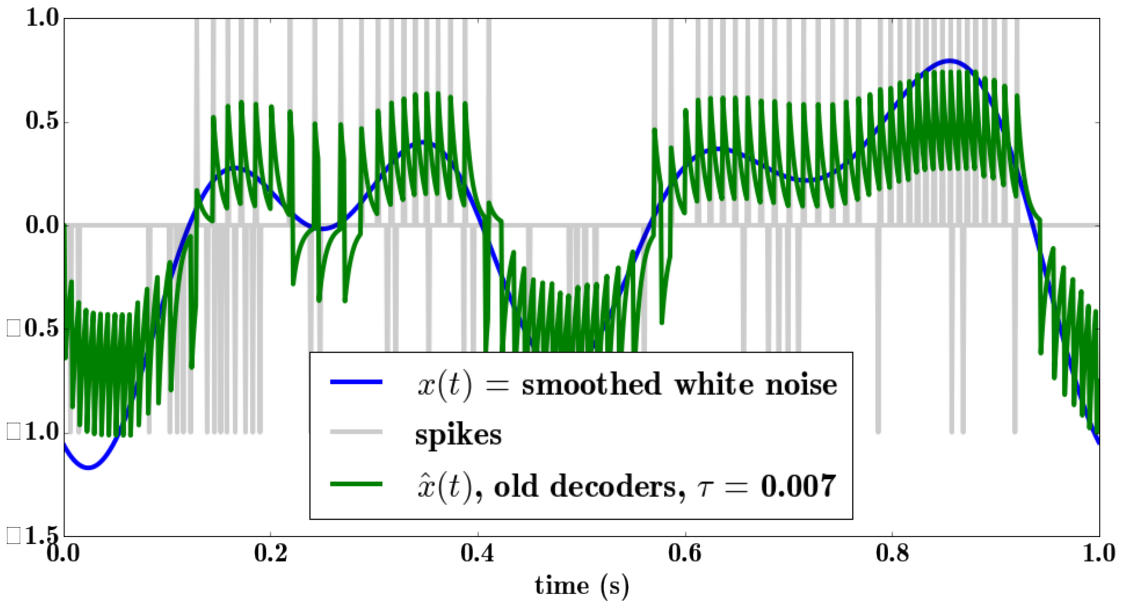
$x(t)$ = smoothed white noise
spikes
$\hat{x}(t)$, $\tau = 0.007$
time (s)



$H(\omega)$
ω



$h(t)$
time (s)



The estimate of the new signal works pretty well with the postsynaptic filter calculated using the old decoders. In both cases the small number of neurons makes it hard to capture the magnitude of the signal or transitions from positive to negative, but the old decoder estimate captures positive/negative values with only a small delay. The fidelity of an estimate calculated with the old decoders to a new signal indicates that postsynaptic filtering can be applied to estimate many signals without needing to fine-tune its parameters (τ), which is advantageous for biological brains conserving mechanisms across diverse cognitive systems. The error is also comparable to that of the optimal filter, without requiring an optimization procedure, indicating it may be a suitable (and biologically motivated) substitute for the optimal filter.