PHY408 Lab 1

Wednesday, February 7, 2024

Jace Alloway - 1006940802 - alloway1

This assignment was compiled in VS Code using the Python and LaTeX extensions. Matplotlib does not produce inline plots in terminal, so the `%matplotlib inline` command was commented out. **Collaborators for all questions: none.**

Problem 1

(a) We begin by convolving two arrays $\{f_i\}$ and $\{w_i\}$ of lengths N_f, N_w respectively into another array $\{g_n\}$. First note that, in array sequences, we index the array beginning at $[0]$ (the first element) up to $[N-1]$ (the last element). We can think of a convolution as mulitplying every element together in both sequences together such that the sum of their indices is the index value of the new array:

$$\begin{aligned} a_0b_0 &\qquad n=0 \\ a_0b_1 + a_1b_0 &\qquad n=1 \\ a_0b_2 + a_1b_1 + a_2b_0 &\qquad n=2 \\ \vdots &\qquad \vdots \end{aligned} \tag{1.1}$$

(I owe credit where credit is due, 3Blue1Brown on YouTube helped explain this: https://www.youtube.com/watch?v=KuXjwB4LzSA) From this we are able to construct a discrete formula as a sum to represent the k-th element of the convolved series:

$$g[k] = \sum_{i,j\,:\,i+j=k} f[i]w[j] \tag{1.2}$$

Under the condition that $i+j=k$, then $j=i-k$ for constant k representing the k-th element. We further note that, by array indexing,

$$0 \leq i \leq N_f - 1 \tag{1.3}$$
$$0 \leq j \leq N_w - 1 \tag{1.4}$$

from which adding both arrays give

$$0 \leq i+j = k \leq N_f + N_w - 2 \tag{1.5}$$

which must have length $(N_f + N_w - 2) + 1 = N_f + N_w - 1$. This is the length of $\{g_n\}$, and we write our sum as

$$g[k] = \sum_{i=0}^{N_f-1} f[i]w[k-i]. \tag{1.6}$$

This mimics the numpy convolve function, but the true convolution must be multiplied by the sampling width Δ:

$$g[k] = \left(\sum_{i=0}^{N_f-1} f[i]w[k-i] \right) \Delta. \tag{1.7}$$

(**b**) We now write the discrete convolution program. For any two arrays a and b which we convolve, taking `len(a) + len(b) - 1` gives the length value of the g array. Writing a for loop for each k value over the convolved array, which indicates the k-th entry of g, then allows us to calculate each term individually (I labelled by terms '`termval`' - which must be reset to 0 after each addition to g).

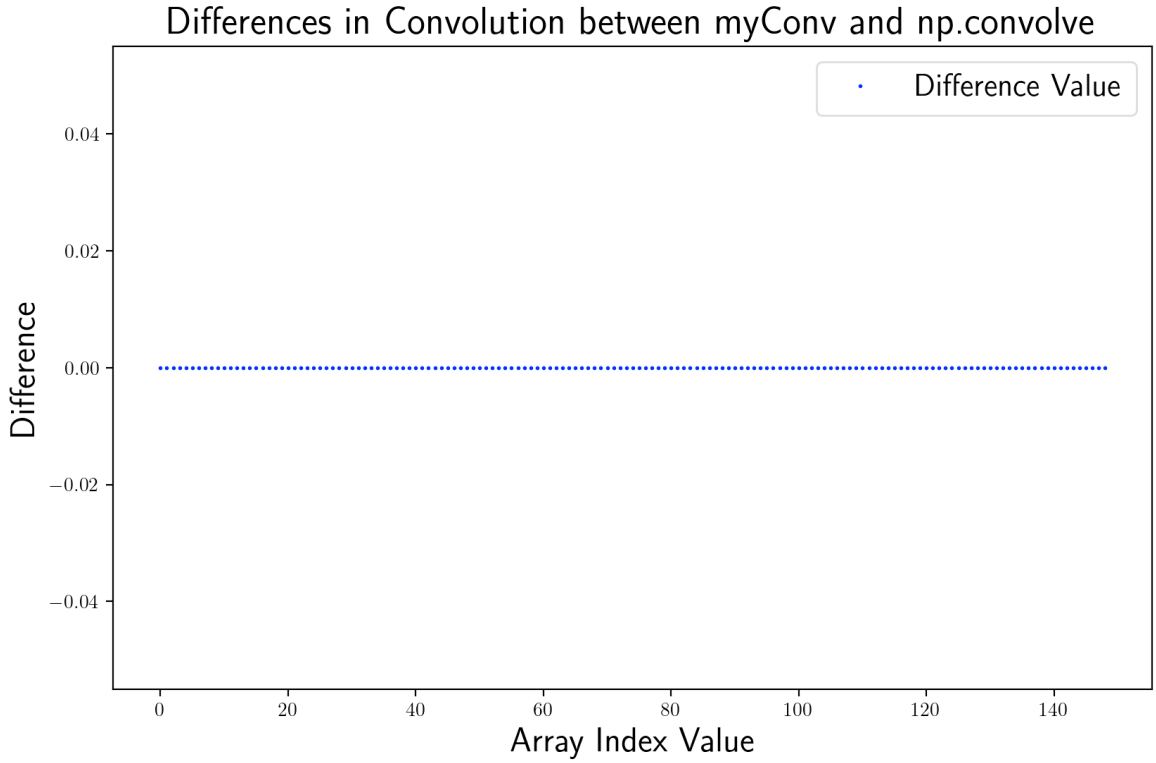
We then loop over the i value from 0 to `len(a)-1`, but we must impose the condition on j for the b array that $0 \leq j = k - i \leq$ `len(b)`-1. If this statement is true, we proceed by calculating the array element `a[i]*b[k-i]` and adding it as the k-th entry of g. We lastly multiply the convolution by the sample width Δ. The function I wrote is then a function of two arrays a and b, and a variable sampling width:

```
def myconv(A, B, DELTA):                #a, b are arrays
    N=len(A)                            #   i index is in 0 and N-1
    M=len(B)                            #   j index is  in 0 and M-1
    K=N+M-1                             #   condition is k=i+j which implies k
                                        #    is in 0 to N+M-2
                                        #   this then implies that  the
                                        #    convolved array length is N+M-1

    conv = np.zeros(K)                  #set length of array to avoid
                                        #appending it each time; make sure
                                        #it stays within the length it needs to
    for k in np.arange(K):
        termval=0                       #reset the value to prevent it
                                        #from adding previous terms
        for i in np.arange(N):
            if 0 <= k-i <= M-1:         #condition on the b[j] index
                termval += A[i] * B[k-i]
        conv[k] = termval*DELTA         #set the convolution array  value

    return conv                         #complete the function
```

(**c**) To test the function, we generate two arrays A and B of lengths 50 and 100, respectively with samples taken from the uniform interval $[0, 1)$ (numpy only allows clopen intervals for some reason). The difference was between my convolution function and the `numpy.convolve()` function was taken out by

```
difference = np.round(myconv(f, w, DELTA) - np.convolve(f,w)*DELTA, 10)
```

where I have rounded the difference value to 10 decimal places. I rounded because, when I had initially ran the code, the difference values I was getting were zero and some non-zero values on the order of 10^{-15}, so since they were so small I attributed it to discrete error and approximated it to be zero by rounding it anyway, since $\mathcal{O}(10^{-15}) \ll 1$. The differences were then plotted (they were all zero):

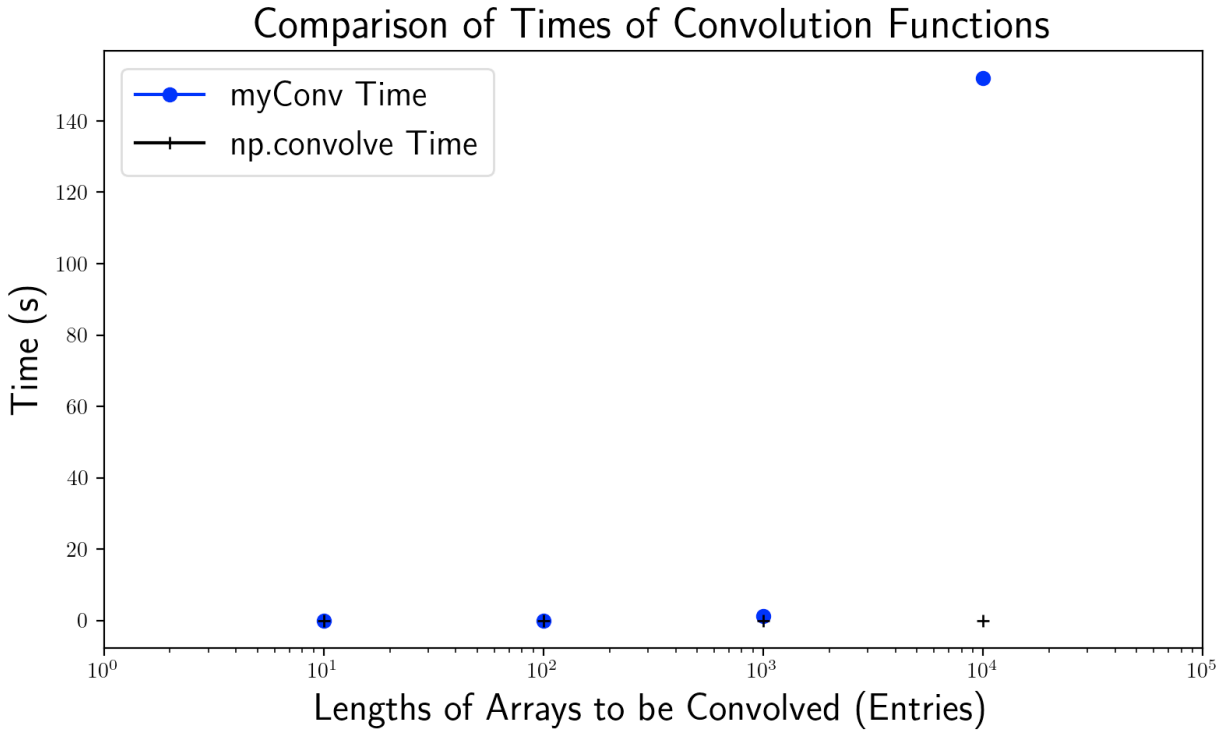


My code is included below:

```
DELTA=1 #fixed sampling interval


#lets generate two random arrays for elements between [0,1]
#of lengths N=50, M=100
N=50
M=100
    #two arrays with uniform distribution between 0 and 1
f = np.random.uniform(0, 1, N)
w = np.random.uniform(0, 1, M)


    #show that my function agrees with np.convolve
difference = np.round(myconv(f, w, DELTA) - np.convolve(f,w)*DELTA, 10)

plt.plot(difference, 'o', color='blue', markersize=1, label='Difference Value')
plt.title('Differences in Convolution between myConv and np.convolve', **tfont)
plt.xlabel('Array Index Value', **afont)
plt.ylabel('Difference', **afont)
plt.legend(loc='best', fontsize=16)
plt.show()
```

where axis and title fonts were previously defined.

(**d**) Using the `time` module, the times of computation were compared between my convolution function as defined in (1.7) and the numpy convolution function.



Above are the plotted times of my function and of `np.convolve()`, where each function took in two randomly generated arrays in $[0, 1)$ of length N, as specified by a lognormal scale on the x-axis. My code for the test is also below.

```
    #proceed by testing the timing values for each array
timing=True            #create toggle (it takes up to 3 minutes :(  )

if timing:
    N = (10, 100, 1000, 10000)      #array of conv lengths
    for i in np.arange(len(N)):         #loop to produce randomly generated
                                        #arrays of length N
        f = np.random.uniform(0, 1, N[i])
        w = np.random.uniform(0, 1, N[i])

        t1=tm.time()
        myconv(f,w, DELTA)     #convolve the arrays and time them
        t2=tm.time()
        np.convolve(f,w)*DELTA
        t3=tm.time()

        print('myconv Time', t2-t1,'Numpy Time', t3-t2, 'for N=%i samples'%N[i])
                                                                    #print the times
```

```
        plt.plot(N[i], t2-t1, color='blue', marker='o', markersize=6)
        plt.plot(N[i], t3-t2, color='k', marker='+', markersize=6)



    plt.plot(-2000,0, label='myConv Time', color='blue', marker='o', markersize=6)
    plt.plot(-2000, 0, label='np.convolve Time', color='k', marker='+', markersize=6)
            #these are just here for the label in legend, since
            #I dont want to keep looping and have 4 labels for the same points
    plt.title('Comparison of Times of Convolution Functions', **tfont)
    plt.xlabel('Lengths of Arrays to be Convolved (Entries)', **afont)
    plt.ylabel('Time (s)', **afont)
    plt.legend(loc='best', fontsize=16)
    plt.xscale('log', base=10)
    plt.xlim([1,10**5])
    plt.show()
```

In all cases, we observe that my function is significantly slower than the numpy convolution, and this is probably because I have explicitly programmed a for loop and sum for each entry (of up to $2N-1$). This means that my function needs to loop for every value, which can take a long time. Numpy is fast because it has siginificant memory and cpu optimizations.

Problem 2

(a) In this problem we study the circuit response of a resistor-inductor loop circuit with an input voltage $V_{\text{in}}(t)$ and an output voltage measured across the inductor $V_L(t)$. We aim to determine $V_L(t)$ as the output response for various $V_{\text{in}}(t)$ functions.

We recall the voltage across a resistor is given by $V_R = IR$ and across an inductor as $V_L = L\dfrac{dI}{dt}$. Kirchoff's loop law implies that we must have

$$V_R(t) + V_L(t) = V_{\text{in}}(t) \tag{2.1}$$

$$\implies I(t)R + L\frac{dI}{dt} = V_{\text{in}}(t). \tag{2.2}$$

We can then solve the ODE for $I(t)$, then take the derivative to obtain the voltage measured across the inductor $L\dfrac{dI}{dt}$ as a function of $V_{\text{in}}(t)$. We have that

$$I\frac{R}{L} + \dot{I} = \frac{1}{L}V_{\text{in}} \tag{2.3}$$

from which the integrating factor is

$$\exp\left[\int dt\, p(t)\right] = \exp\left[\int dt\, \frac{R}{L}\right] \tag{2.4}$$

$$= e^{Rt/L}. \tag{2.5}$$

This then yields that

$$\frac{d}{dt}\left[e^{Rt/L}I\right] = \frac{1}{L}e^{Rt/L}V_{\text{in}} \tag{2.6}$$

Integrating from some arbitrary initial time $-\infty$ to t then gives the solution to $I(t)$ as a function of $V_{\text{in}}(t)$:

$$I(t) = \frac{1}{L}e^{-Rt/L}\int_{-\infty}^{t} dt' e^{Rt'/L}V_{\text{in}}(t') \tag{2.7}$$

The definition of the output voltage response is $V_L(t) = L\dfrac{dI}{dt}$, so taking the derivative implies

$$V_L(t) = \frac{d}{dt}\left[e^{-Rt/L}\int_{-\infty}^{t} dt' e^{Rt'/L}V_{\text{in}}(t')\right] \tag{2.8}$$

$$= -\frac{R}{L}e^{-Rt/L}\int_{-\infty}^{t} dt' e^{Rt'/L}V_{\text{in}}(t') + e^{-Rt/L}e^{Rt/L}V_{\text{in}}(t) \tag{2.9}$$

$$= V_{\text{in}}(t) - \frac{R}{L}\int_{-\infty}^{t} dt' e^{R/L(t'-t)}V_{\text{in}}(t') \tag{2.10}$$

$$= V_{\text{in}}(t) - \frac{R}{L}\left(e^{Rt/L} * V_{\text{in}}(t)\right). \tag{2.11}$$

which is a convolution. First suppose that $V_{\text{in}}(t) = H(t)$, the Heaviside step function. For $t < 0$, everything is zero. For $t > 0$, we obtain that

$$V_L(t) = 1 - \frac{R}{L}\int_{0}^{t} dt' e^{R/L(t'-t)} \tag{2.12}$$

$$= 1 - \frac{R}{L}\frac{L}{R}e^{-Rt/L}\left[e^{Rt'/L}\right]_0^{t'} \tag{2.13}$$

$$= 1 - 1 + e^{-Rt/L} \tag{2.14}$$

$$= e^{-Rt/L} \tag{2.15}$$

which must be only true for $t > 0$, so we may also write

$$V_L(t) = e^{-Rt/L}H(t) \tag{2.16}$$

as desired. Now, if $V_{\text{in}}(t) = \delta(t)$, we have that

$$V_L(t) = \delta(t) - \frac{R}{L}\int_{-\infty}^{t} dt'\, e^{R/L(t'-t)}\delta(t') \tag{2.17}$$

$$= \delta(t) - \frac{R}{L}e^{-Rt/L}\int_{-\infty}^{t} dt' 1 \cdot \delta(t') \tag{2.18}$$

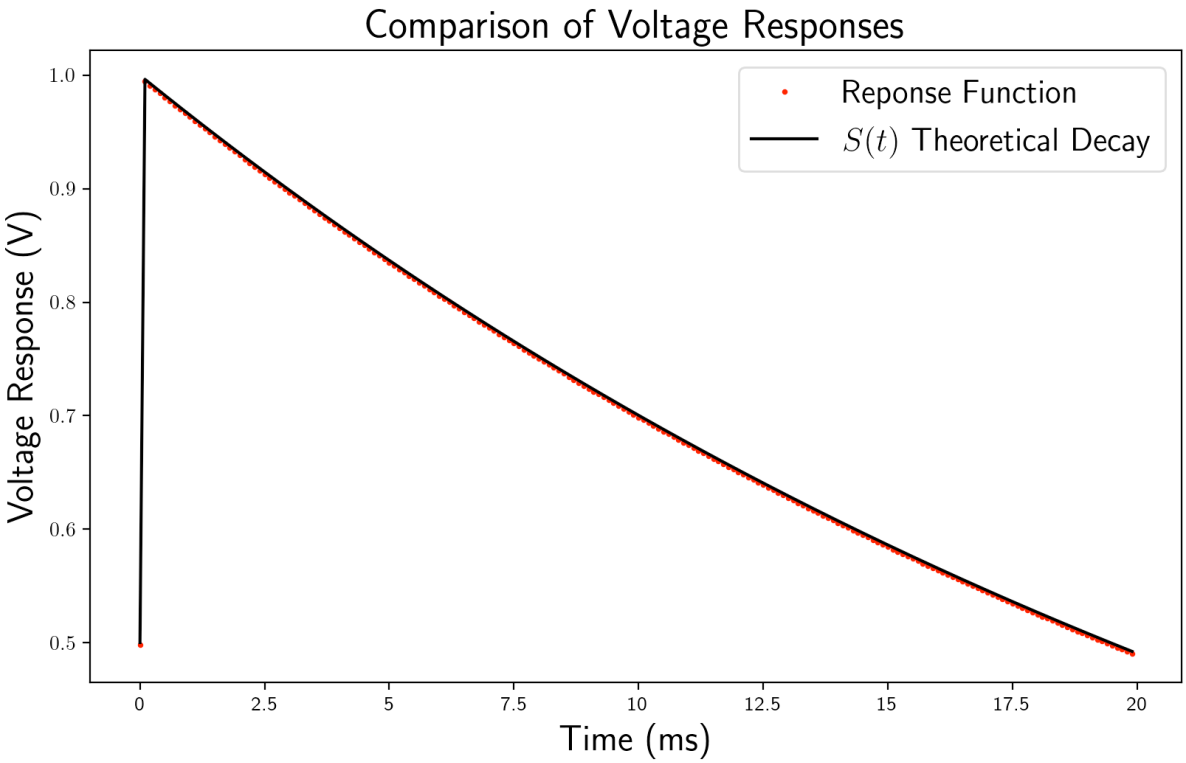
$$= \delta(t) - \frac{R}{L}e^{-Rt/L}H(t) \tag{2.19}$$

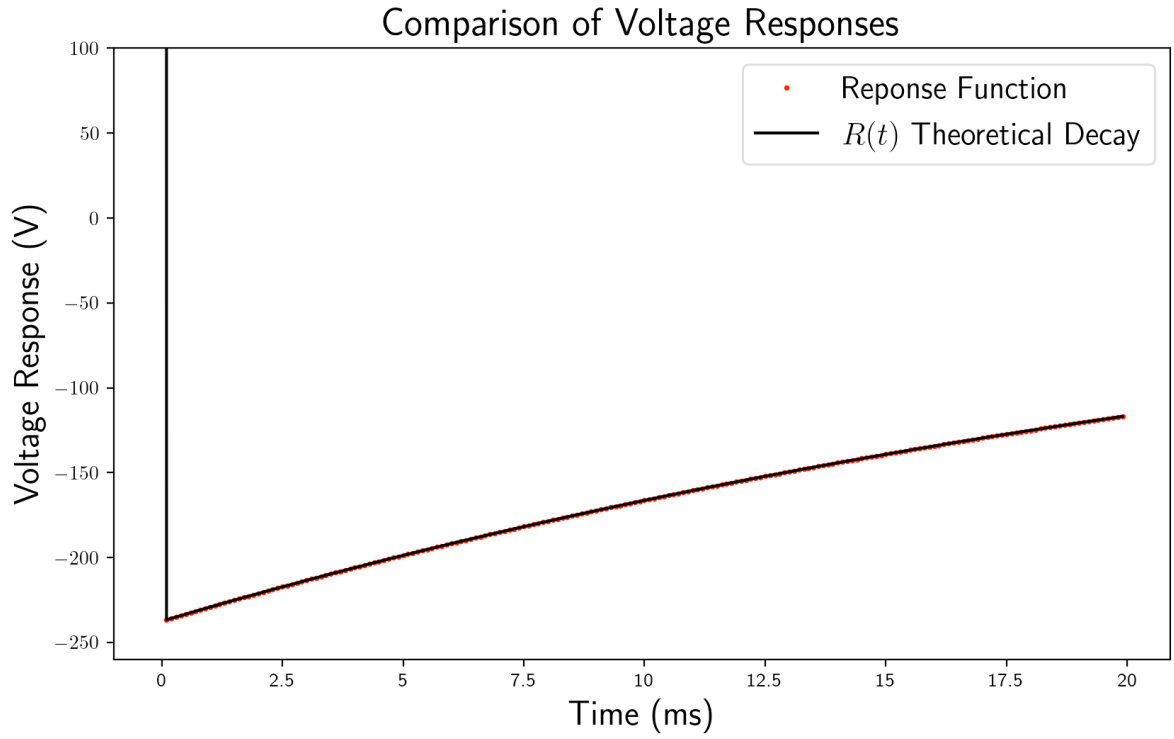
where the last line follows from the fact that $\frac{d}{dt}H(t) = \delta(t)$ by the fundamental theorem of calculus, so we have $\frac{d}{dt}\int_{-\infty}^{t} dt'\,\delta(t') = \delta(t)$.

(**b**) We begin by discretizing the input voltages. The step function was done by defining a time array of variable length and sampling width dt, then using `np.ones(len(time))` to create an array of ones. Then I appended `H[0] = 0.5`. Similarly, for the delta function, $\delta(t)$ was created using `np.zeros(len(time))` and `D[0] = 1/dt`. The $S(t)$ and $R(t)$ impulses arrays were also defined as functions. The response function was defined exactly as in (2.11), with the addition of a slice of lenth `[0, len(V_in)]` so that the operands could be broadcasted together with the same shape. My function is pasted below, where the 'time' array is assumed to already be specified:

```
def RLresponse(R, L, V_in, dt):              #define time evolution;
                                             #time should be previously defined by
                                             #specifying the lengths of H, D
    conv = np.convolve(np.exp(-R*time/L), V_in)*dt
    V_out = V_in - (R/L)*conv [0:len(V_in)]
    return V_out
```

(**c**) As a test, we take $R = 950\Omega$ and $L = 4H$ with a sampling period of $dt = 0.15$ms (that is, 0.000015s). I have only included the first 20ms of data, and this was done by just slicing the convolved array (since the sample number changed) from `[0:200]`. The plots for the discretized input voltage arrays (2.16) and (2.19) are below:





The python code of the testing and plotting is also included below:

```
R=950
L=4
```

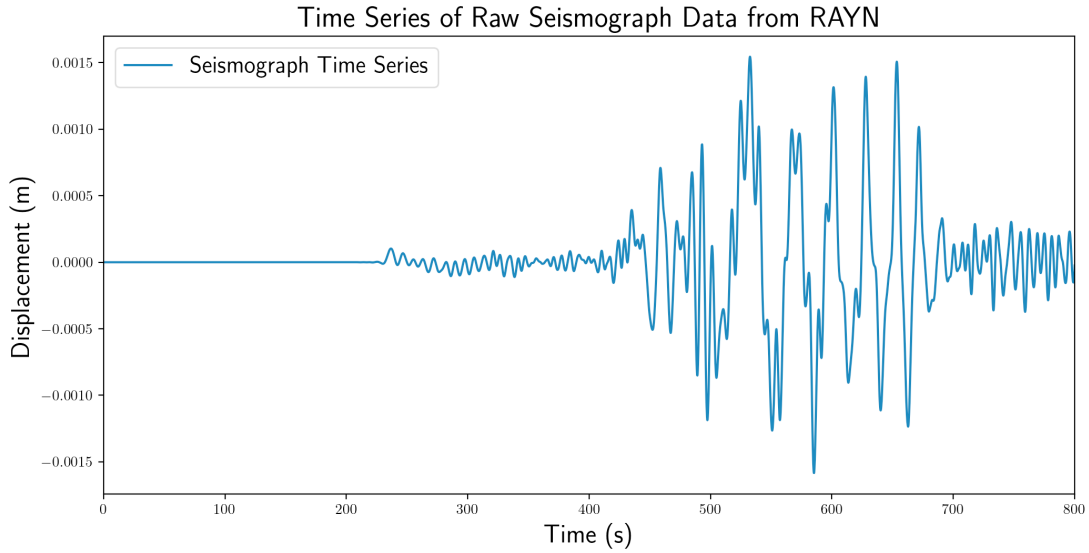
```
    #plot for H_n
fig, ax=plt.subplots(1,1)
ax.plot(RLresponse(R, L, H, dt)[0:200], 'o', color='red', markersize=1.7,
                                                    label='Reponse Function')
ax.plot((H*np.exp(-R*time/L))[0:200], color='k', lw=1.5,
                                                    label=r'$S(t)$ Theoretical Decay')
ax.legend(loc='best', fontsize=16)
ax.set_xlabel('Time (ms)', **afont)
ax.set_xticklabels([0, 0, 2.5, 5, 7.5, 10, 12.5, 15, 17.5, 20])
ax.set_ylabel('Voltage Response (V)', **afont)
ax.set_title('Comparison of Voltage Responses', **tfont)
plt.show()

    #plot of D_n
fig, ax=plt.subplots(1,1)
ax.plot(RLresponse(R, L, D, dt)[0:200], 'o', color='red', markersize=1.7,
                                                    label='Reponse Function')
ax.plot((impulse(R, L))[0:200], color='k', lw=1.5,
                                                    label=r'$R(t)$ Theoretical Decay')
ax.legend(loc='best', fontsize=16)
ax.set_xlabel('Time (ms)', **afont)
ax.set_xticklabels([0, 0, 2.5, 5, 7.5, 10, 12.5, 15, 17.5, 20])
ax.set_ylabel('Voltage Response (V)', **afont)
ax.set_title('Comparison of Voltage Responses', **tfont)
ax.set_ylim(-260, 100)
plt.show()
```

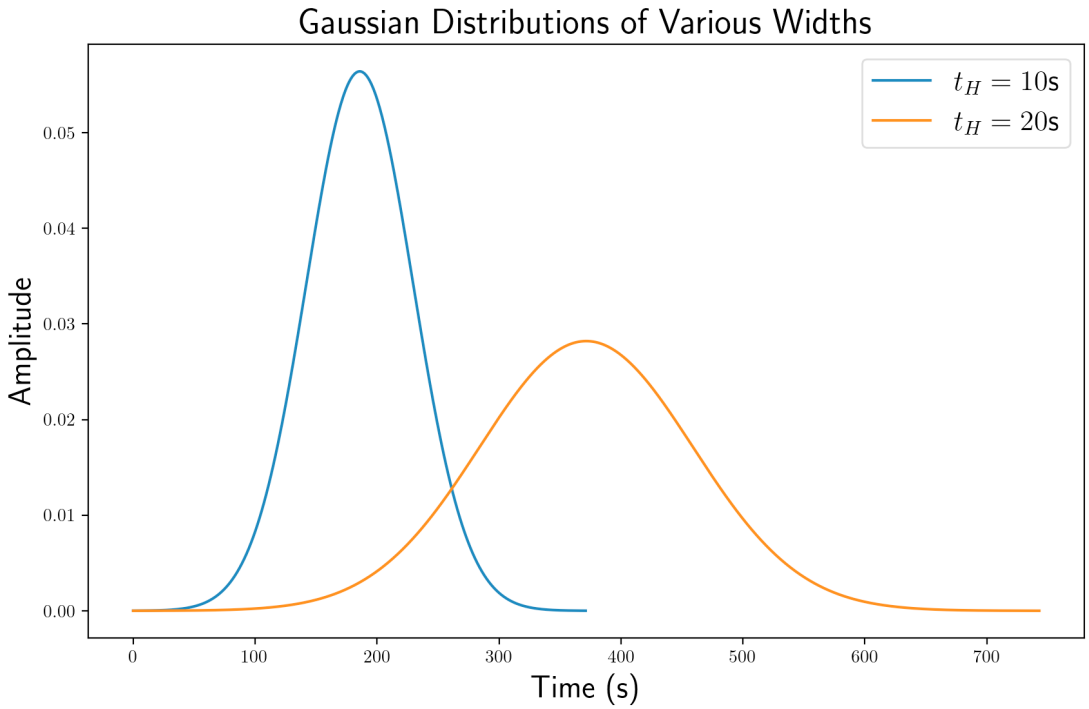
Problem 3

(**a**) This problem focuses on the convolution of synthetic seismograms. The seismogram data was loaded using `np.loadtxt()` with `unpack=True`, which allows me to isolate the temporal and displacement arrays of the data. The timestep of the seismogram was assumed to be constant for each sample, so I took the sampling value to be `dt = semtime[10] - semtime[9]` arbitrarily. The Gaussian function was then defined as

$$g(t) = \frac{1}{\sqrt{\pi} t_H} e^{-(t/t_H)^2} \tag{3.1}$$

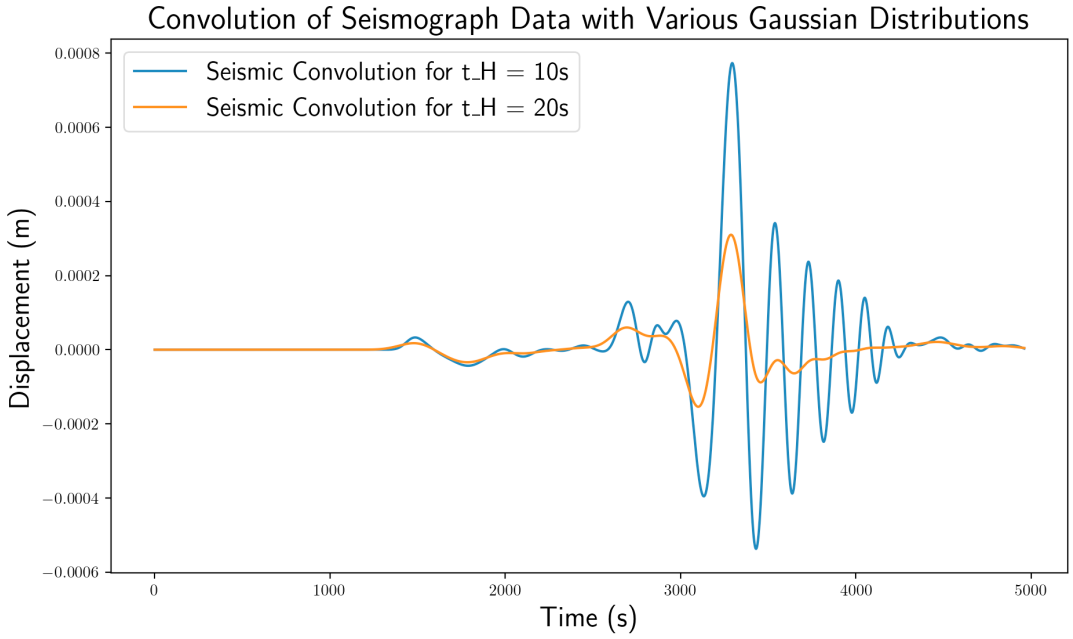
and plotted in a loop for various $t_H = (10, 20)$ values.





The plots for the raw seismogram data and both Gaussians (overlayed) are included above (they won't have the same center or width because t_H is different for both).

(**b**) Lastly, using `np.convolve`, the raw data from the seismogram was convolved with the arrays of the Gaussian distributions for both $t_H = 10$s and $t_H = 20$s. The below plot includes both overlayed:



Observe that the Gaussian function acts as a 'smoothing' operation in convolution with the raw data, and the wider the width of the Gaussian the more prevalent its effect will be on the raw data. We not only observe it smoothing the data, but also decreasing it's amplitude, and this is because as time proceeds in both arrays, the Gaussian eventually tends to zero. This implies that for large t_H values and long times, the raw data would become zero (a flat line). This is commonly seen in partial differential equations, one instance being the heat equation, as the input heat distribution is convolved with an exponential in time to reduce the heat to equilibrium over time.