DSP Lab 1

Exe 2.2

The delta function was implemented as shown in figure 1.

```
function seq = delta(n)
    seq = n == 0;
end
```

Figure 1 - Delta Function Code

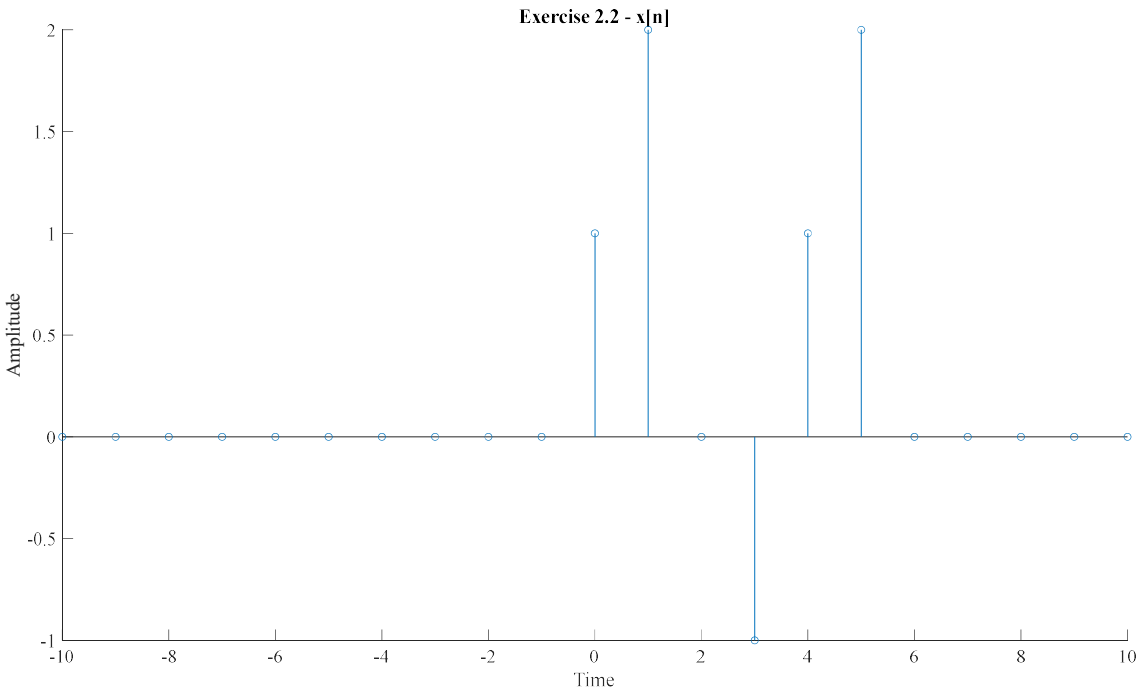


Figure 2 - x[n] from -10 to 10

Exe 2.3

This part of the lab had us delay the sequence with various time lengths. This was done using the “delayseq” function. This accepts the sequences as a column vector as well as the number of samples to delay by. Since we sample every second, the delay length was used as the number of samples to delay by. The graph of the delayed functions can be seen in figure 3.

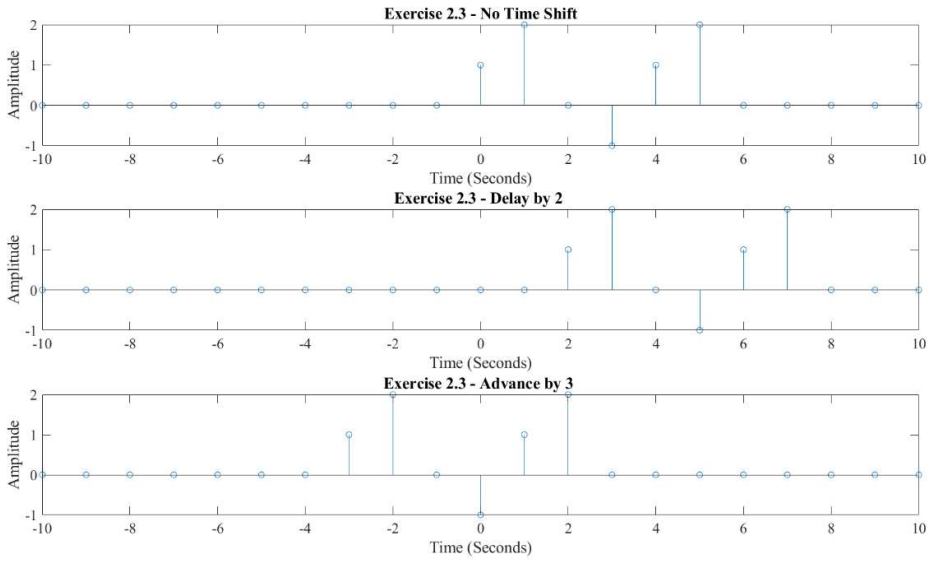


Figure 3 - Graph of delayed sequences

It should be noted that the time axis for the second and third sequences is based on the first sequence.

Exe 2.4

In order to align the sequences, I had to make new function called alignSeq. This accepts two sequences and their respective time arrays and creates a new array based on the max and min time of both time arrays. Using this new array called nOut, the function calculates the relevant number of zeros to add to the array so that both sequences are properly time aligned. The function is shown below in figure 4. With the time aligned sequences in figure 5.

```
function [y1, y2, nOut] = alignSeq(x1, t1, x2, t2)

    nOut = min(t1(1), t2(1)):1:max(t1(length(t1)), t2(length(t2)));
    x1 = horzcat(x1, zeros(1,length(nOut) - length(x1)));
    x1 = circshift(x1, find(nOut == t1(1)) - 1);

    x2 = horzcat(x2, zeros(1,length(nOut) - length(x2)));
    x2 = circshift(x2, find(nOut == t2(1)) - 1);

    y1 = x1;
    y2 = x2;

end
```

Figure 4 - alignSeq function used to align sequences

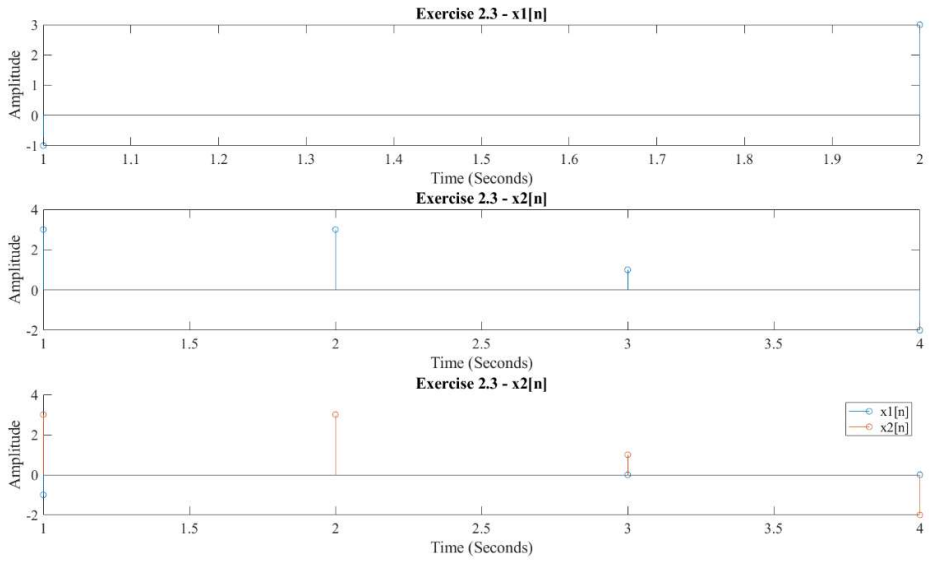


Figure 5 - Graph showing aligned x1[n] and x2[n]

However, it should be noted that this function only works if the time step for each sequence is 1. In later exercises, this function was further improved to take in a variable step size.

Exe 2.5

This was a simple extension of the function developed in the previous exercise. Instead of returning the aligned sequences, the function (now renamed to addSeq) returns the sum of the two input sequences. The complexity in this task comes from the time alignment of the signals rather than the signal amplitude. So, to demonstrate the functionality of the function I came up with a sequence $x_1[n]$ ranging from $t \in [-10, 3]$ and a sequence $x_2[n]$ ranging from $t \in [-3, 13]$. Their amplitudes were set to range from 1 to the length of the array with a step size of 1. The summation of the time aligned sequences can be seen in figure 6.

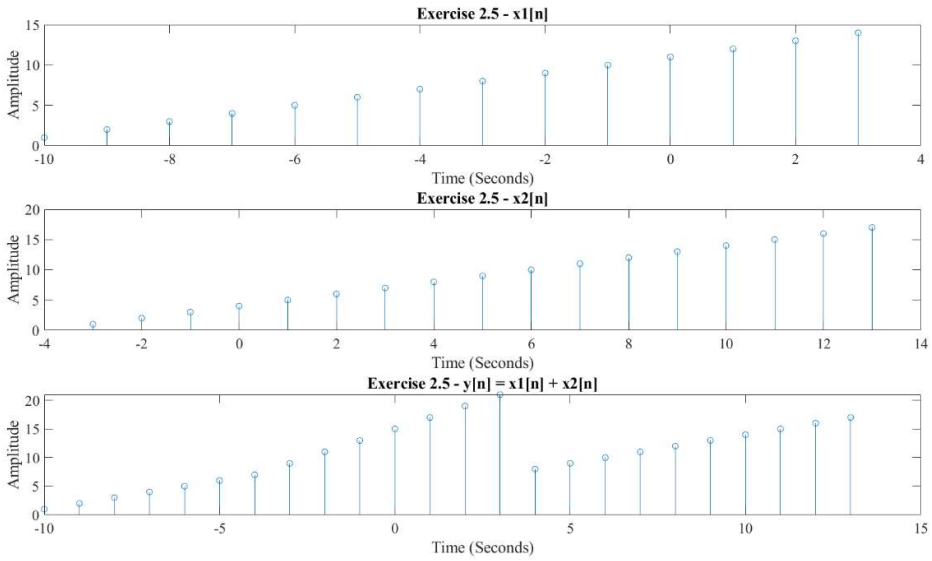


Figure 6 - Summation of time aligned sequences

Exe 2.6

We were given the sequence x[n] defined as:

$$x[n] = \delta(t+3) + \delta(t+1) + \delta(t-3)$$

$h[n]$ is defined as:

$$h[n] = \begin{cases} 1 \text{ n} = \alpha \\ 0 \text{ otherwise} \end{cases}$$

We can take the convolution of the two sequences using the Laplacian property of convolutions which states:

$$\mathcal{L}\big((f*g)(t)\big) = \mathcal{L}\big(f(t)\big)\mathcal{L}\big(g(t)\big)$$

$$\mathcal{L}(x[n]) = e^{3s} + e^{s} + e^{-3s}$$

$$\mathcal{L}(h[n]) = e^{-\alpha s}$$

$$\mathcal{L}\big((f*g)(t)\big) = [e^{3s} + e^{s} + e^{-3s}]e^{-\alpha s}$$

$$\mathcal{L}\big((f*g)(t)\big) = e^{-(\alpha-3)s} + e^{-(\alpha-1)} + e^{-(3+\alpha)s}$$

$$y[n] = \delta(t-\alpha-3) + \delta(t-\alpha+1) + \delta(t-3-\alpha)$$

In essence it is just time shifting the original sequence by α. This can be seen in figure 7 where $\alpha = 1$.

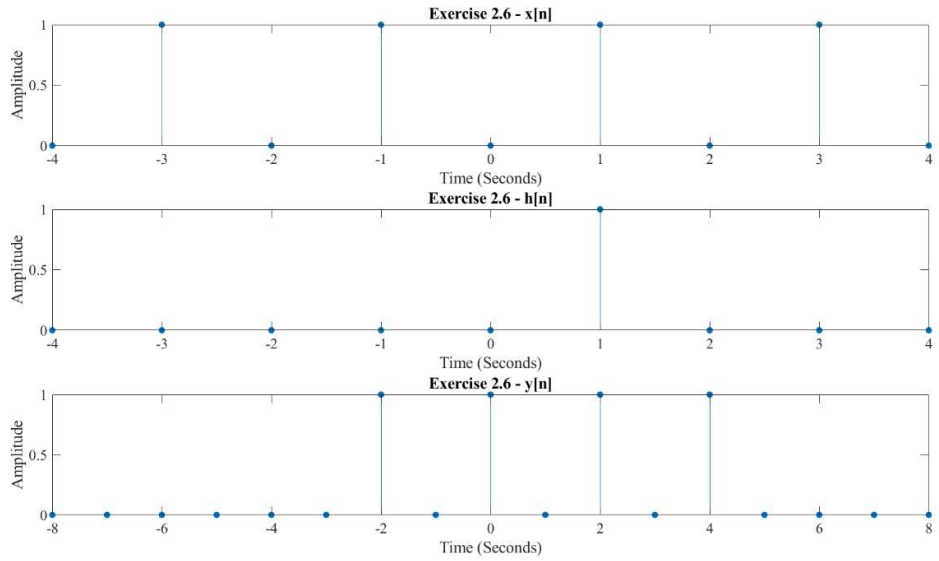
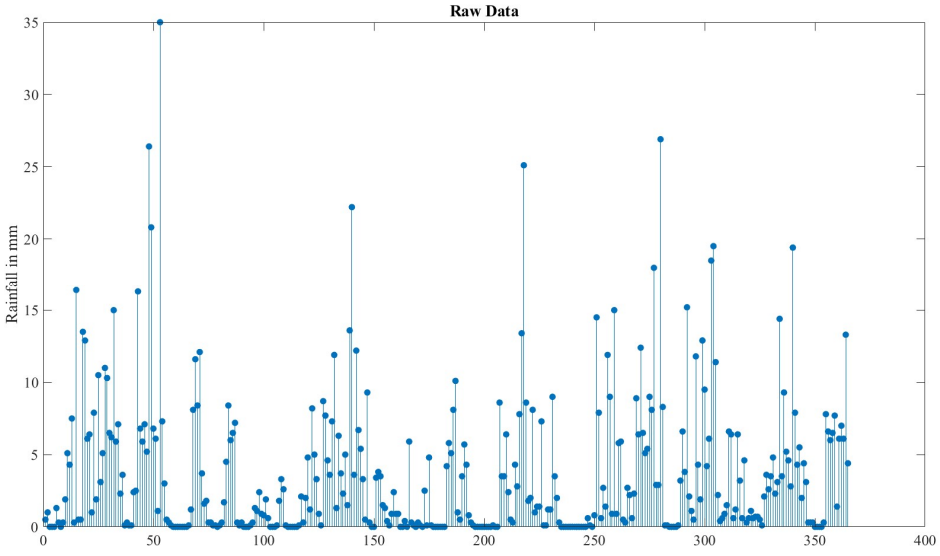
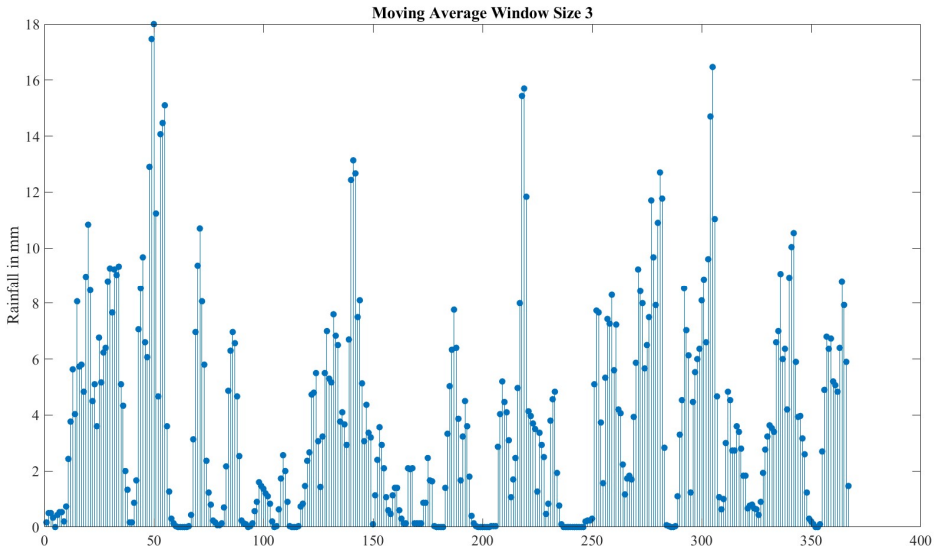


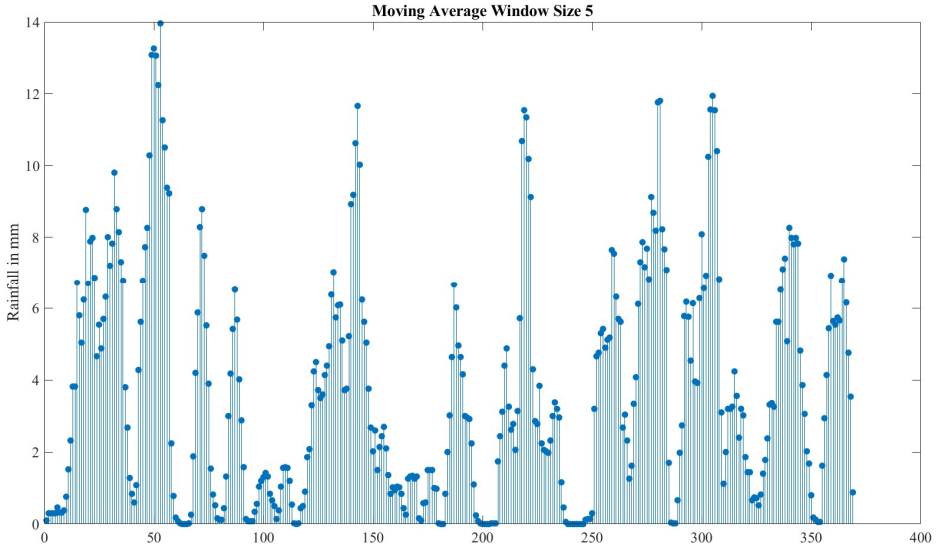
Figure 7 - Convolution of x[n] and h[n]

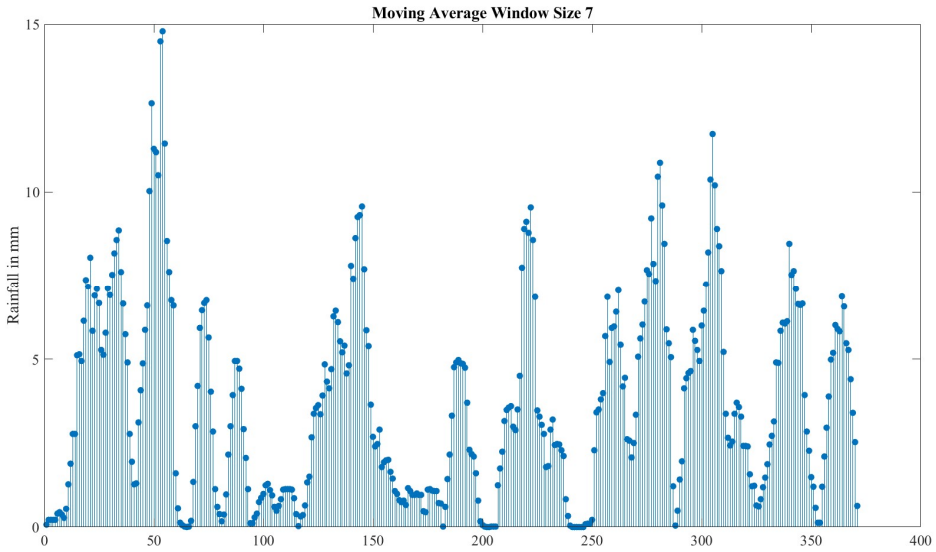
Exe 2.7

A moving average filter was implemented by first creating an array of length WindowSize whose coefficients were set to be 1/WindowSize. We then convolve the array with the sequence that needs to averaged. I tried various window sizes from 3 – 10 and displayed the results in figure 8.




Moving Average Window Size 3
Rainfall in mm
0
2
4
6
8
10
12
14
16
18
0
50
100
150
200
250
300
350
400


Moving Average Window Size 5
Rainfall in mm
0
2
4
6
8
10
12
14
0
50
100
150
200
250
300
350
400



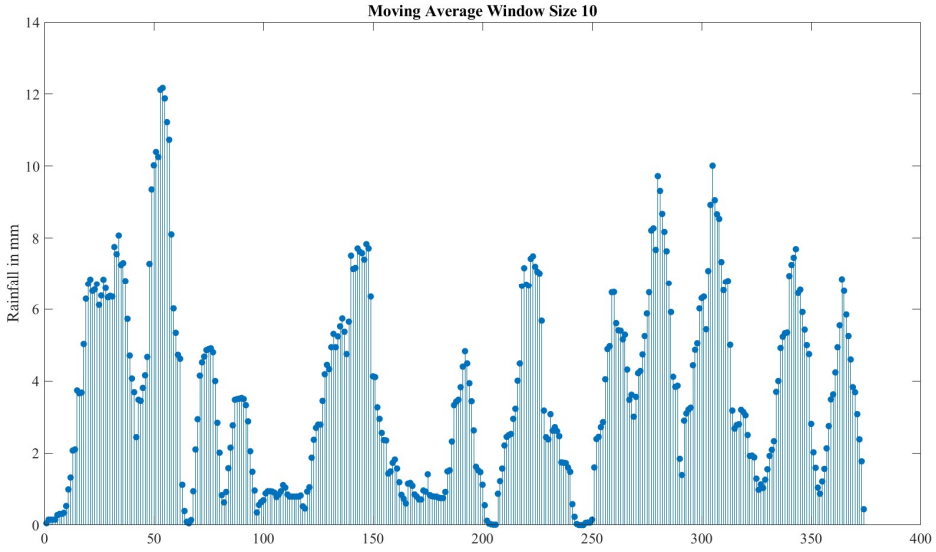


Figure 8 - Various window sizes for the moving average filter

From the graphs above it seems that high values for $h[n]$ seem to have very little effect on the smoothness of the data, however smaller values (corresponding to a longer window) then you get a smoother result. A window size of 5 – 7 seems to give a good result.

Exe 4.2

The relationship between f_s, N and d is given by:

$$N = f_s \times d + 1$$

Exe 4.5

The value of f_0 controls the pitch of the sine wave. Higher value of f_0 results in a higher pitched sound, while lower values leads to a bass-ier lower pitched sound.

Exe 4.6

The fundamental period of $x(t)$ before sampling is given by:

$$T_0 = \frac{1}{f_0}$$

In this case a frequency of 300Hz was chosen leading to a fundamental period of :

$$T_0 = \frac{1}{300} = 3.33 \times 10^{-3} \text{ seconds}$$

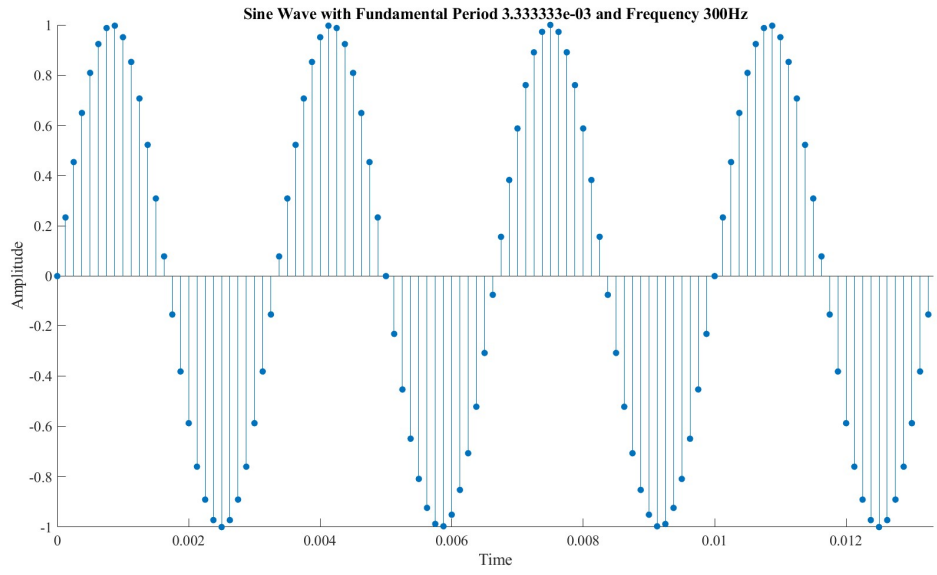


Figure 9 - Sine Wave with a frequency of 300Hz

Figure 9 shows a plot of a sine wave of 300Hz between times $t = 0$ and $t = 4T_0$.

Exe 4.7

Here we had to delay the sequence by two seconds. This was done using the delayseq function. The results are shown in figure 10.

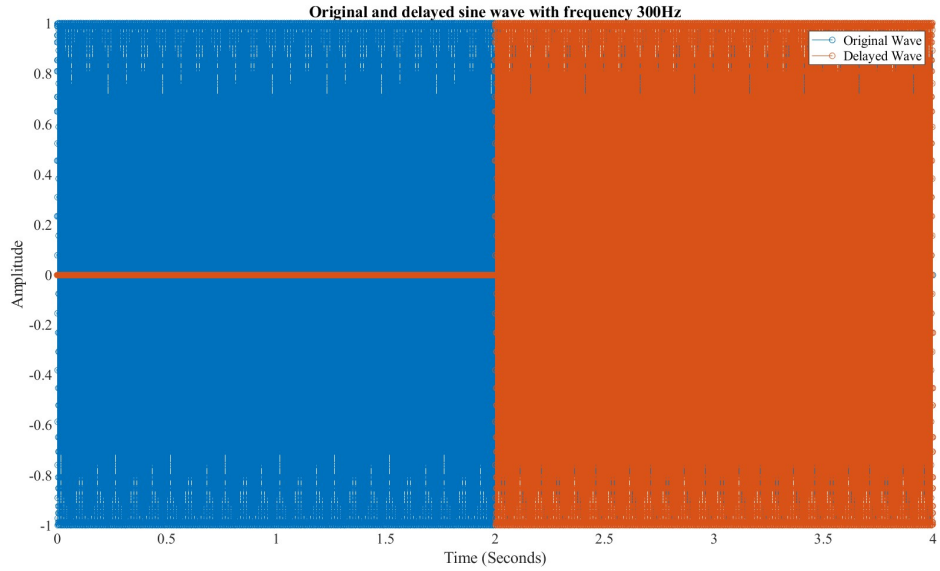
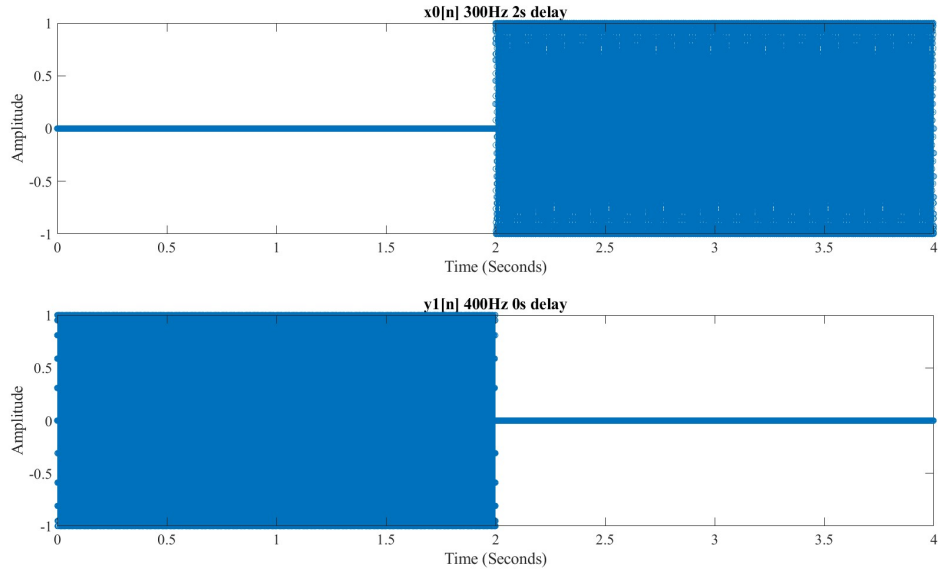


Figure 10 - Original and delayed sine waves

Exe 4.8

This exercise required us to create various sine waves with different offsets and add them to the original signal $x_0[n]$ with a frequency of 300Hz and a delay of 2 seconds. The time aligned (in relation to $x_0[n]$) are shown in figure 11.



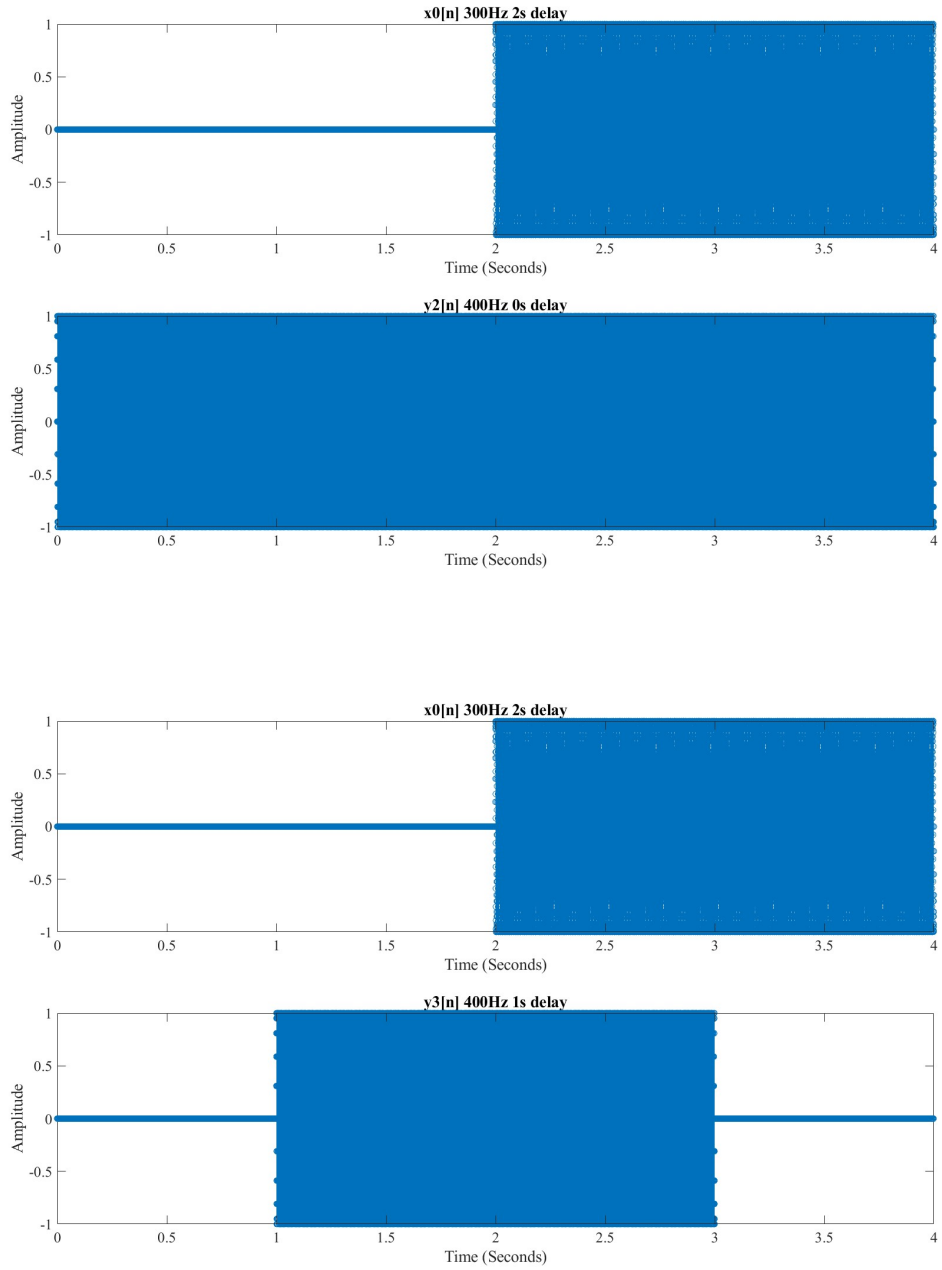


Figure 11 - Time aligned sine waves

All these sine waves were summed to $x_0[n]$, a time domain graph is not useful in this case since it is not evident that the resulting sine wave is any different than the original. So to show that the addition was successful and done in a time aligned manner, the FFT of the resulting sequence was taken and is displayed in figure 12.

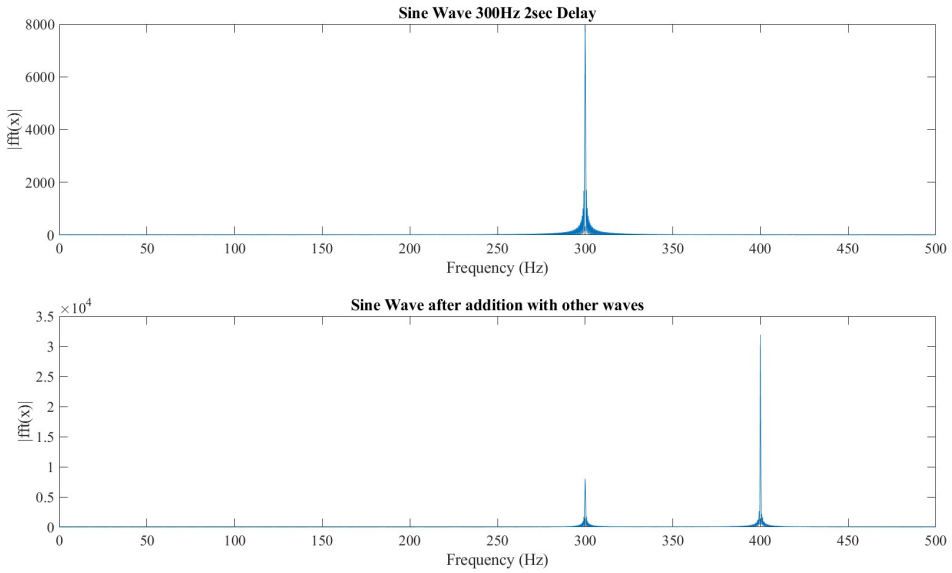


Figure 12 - x0[n] on top with the sequence after addition shown below

Exe 4.9

This exercise had us implement a function that would accept two sequences and their time arrays, align time and either add or multiply them depending on what was needed. The function is implemented as shown in figure 13.

```
function [y, nOut] = addMulSeq(x1, t1, x2, t2, stepSize, addMul)
    nOut = min(t1(1), t2(1)):stepSize:max(t1(length(t1)), t2(length(t2)));
    x1 = horzcat(x1, zeros(1,length(nOut) - length(x1)));
    x1 = circshift(x1, find(nOut == t1(1)) - 1);

    x2 = horzcat(x2, zeros(1,length(nOut) - length(x2)));
    x2 = circshift(x2, find(nOut == t2(1)) - 1);

    if(addMul == 0)
        y = x1+x2;
    elseif(addMul == 1)
        y = x1.*x2;
    else
        error("Incorrect value %d for addMul option\n", addMul);
    end
end
```

Figure 13 - Implementation of addMulSeq

Exe 4.10

This exercise required us to create a function that accepted two sequences, their respective time arrays and calculated the convolution between both sequences. This was implemented using the function seen in figure 14.

```
function [y, nOut] = convSeq(x1, t1, x2, t2, stepSize)
    y = conv(x1, x2);
    nOut = t1(1) + t2(1):stepSize:(t1(1) + t2(1) + length(y) - 1)*stepSize;
end
```

Figure 14 - Implementation of the convSeq function

Once this was implemented, I passed in the sequences defined Exe 4.8 and computed successive convolutions, i.e. I passed the output of the convolution with the previous sequence as the first input with the next sequence. The result of these convolutions is shown below in figure 15.

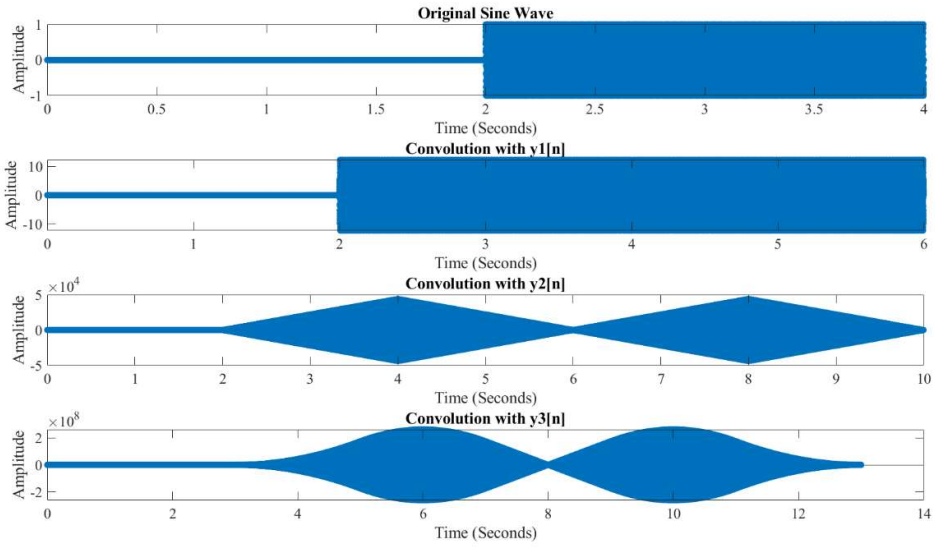


Figure 15 - Result of the convolutions